NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

MISSION PLANNING AND MISSION CONTROL SOFTWARE FOR THE PHOENIX AUTONOMOUS UNDERWATER VEHICLE (AUV): IMPLEMENTATION AND EXPERIMENTAL STUDY

by

Bradley J. Leonhardt

March, 1996

Thesis Advisors: Donald P. Brutzman
Robert B. McGhee

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE March 1996	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE MISSION PLANNING AND MISSION CONTROL SOFTWARE FOR THE PHOENIX AUTONOMOUS UNDERWATER VEHICLE (AUV): IMPLEMENTATION AND EXPERIMENTAL STUDY	5. FUNDING NUMBERS
6. AUTHOR(S) Bradley J. Leonhardt	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

The Naval Postgraduate School Autonomous Underwater Vehicle (AUV), Phoenix, has a well developed lower level architecture (Execution level) while the upper, Strategic and especially the Tactical, levels need refinement. To be useful in the fleet an easier means of creating mission code for the Strategic level is required. A software architecture needed to be implemented at the Tactical level on-board Phoenix which can accommodate multi-processes, multi-languages, multi-processors and control hard real-time constraints existing at the Execution level. Phoenix also did not have a path replanning capability prior to this thesis.

The approach taken is to provide Phoenix a user-friendly interface for the autogeneration of human-readable mission code and the creation and implementation of a Tactical level control architecture on-board Phoenix to include path replanning. The approach utilizes Rational Behavior Model (RBM) architectural design principles. This thesis focuses on the Officer of the Deck and replanning at the Tactical level, and refinement of the Captain at the Strategic level. While further testing is necessary, Phoenix is now capable of behaving as a truly autonomous vehicle.

Results of this thesis show that nontechnical personnel can generate Prolog code to perform missions on-board Phoenix. Path replanning and obstacle avoidance software are also implemented. Most important this thesis demonstrates successful operation of all three levels of the RBM architecture on-board Phoenix.

14. SUBJECT TERMS Path planning Robotic Architecture Expert System	15. NUMBER OF PAGES 253
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18 298-102

MISSION PLANNING AND MISSION CONTROL SOFTWARE FOR THE PHOENIX AUTONOMOUS UNDERWATER VEHICLE (AUV): IMPLEMENTATION AND EXPERIMENTAL STUDY

Bradley J. Leonhardt
Lieutenant, United States Navy
B.S., Auburn University, 1989

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1996

Author: Bradley J. Leonhardt
Bradley J. Leonhardt

Approved by: [illegible]
Donald P. Brutzman, Thesis Co-Advisor

[illegible]
Robert B. McGhee, Thesis Co-Advisor

[illegible]
Ted Lewis, Chairman
Department of Computer Science

ABSTRACT

The Naval Postgraduate School Autonomous Underwater Vehicle (AUV), Phoenix, has a well developed lower level architecture (Execution level) while the upper, Strategic and especially the Tactical, levels need refinement. To be useful in the fleet an easier means of creating mission code for the Strategic level is required. A software architecture needed to be implemented at the Tactical level on-board Phoenix which can accommodate multi-processes, multi-languages, multi-processors and control hard real-time constraints existing at the Execution level. Phoenix also did not have a path replanning capability prior to this thesis.

The problems addressed by this thesis include providing Phoenix a user-friendly interface for the autogeneration of human-readable mission code and the creation and implementation of a Tactical level control architecture on-board Phoenix to include path replanning. The approach taken utilizes Rational Behavior Model (RBM) architectural design principles. This thesis focuses on the Officer of the Deck and replanning at the Tactical level, and refinement of the Captain at the Strategic level. While further testing is necessary, Phoenix is now capable of behaving as a truly autonomous vehicle.

Results of this thesis show that nontechnical personnel can generate Prolog code to perform missions on-board Phoenix. Path replanning and obstacle avoidance software are also implemented. Most important this thesis demonstrates successful operation of all three levels of the RBM architecture on-board Phoenix.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

I remember well the day Dr. Don Brutzman spoke to our class about his research. It was his turn in the parade of professors to present their areas of interest in hopes of obtaining thesis students. His theme was simple: "Do real work." He further elaborated on this concept by asking each of us to think about what we will say to our colleagues later when asked what we did at the Naval Postgraduate School. I then remembered a course I had with Dr. Robert McGhee. He showed us a video tape of an autonomous vehicle the school had. It was inside a 20 by 20 foot square, 6 foot deep pool. It had a cable coming out of it to pool-side computers. Professor McGhee spoke about the many areas of research that still needed to be accomplished with this "autonomous vehicle."

I would like to thank Professor McGhee for making learning Artificial Intelligence interesting and practical. A big thanks goes to Professor Don Brutzman who has spent many a late night working with me and the project to advance it to its current stage.

Without the support of my family, who put up with my late night and weekend absences, I would not have succeeded. Knowing when to say when is an impossible task when evolved in such a worthwhile project. Leaving the project after graduation is much like leaving a ship, with all the emotions and satisfaction from accomplishing a job and leaving it in better shape then when you started.

This research was supported in part by Grant BCS - 9306252 from the National Science Foundation to the Naval Postgraduate School.

I. INTRODUCTION

A. MOTIVATION

The Naval Postgraduate School in Monterey California, is actively involved in autonomous underwater vehicle (AUV) research. It has established a Center for Autonomous Underwater Vehicle Research (CAUVR) that is exploring many concepts in the design and control of AUVs. The concepts developed have been applied to the NPS Phoenix AUV. Many years have gone into the creation of a stable platform for Phoenix at the motor/controller level. Phoenix has been in testing at the CAUVR test tank connected to external computers that provide computer support for its operation (Figure 1). Of course if this vehicle is to ever be considered truly autonomous, it needs to be able to run all its software from within the physical confines of the vehicle.

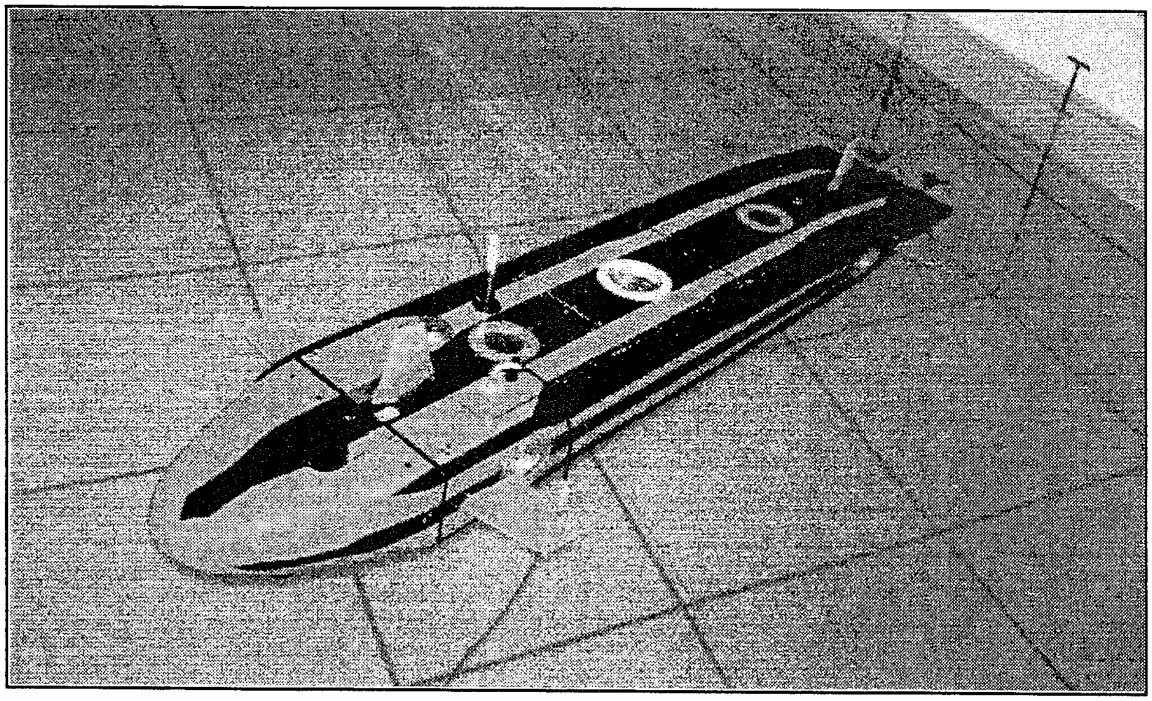


Figure 1. Phoenix AUV undergoing testing at the Center for AUV Research (CAUVR) lab test tank in early 1995. Monitoring of operation via pool side computers during code development was provided via an external Ethernet tether.

1. Mine Warfare

One area of research this vehicle is investigating is mine clearance in the shallow water zone (Figure 2). The Navy has a last-ditch method for hunting mines, and it's not high-tech, glamorous or foolproof. It is a gunner's mate on the bow of a ship, armed with an M-14 rifle [Boorda 96]. For a Navy that wants to reduce the mine threat to no more than a "speed bump," sailors sniping at floating mines will no longer be adequate. Mines in the Persian Gulf War caught the Navy and Marines off guard. In fact, mines there derailed plans for a Marine landing on the coast of Kuwait. Navy planners vowed it will

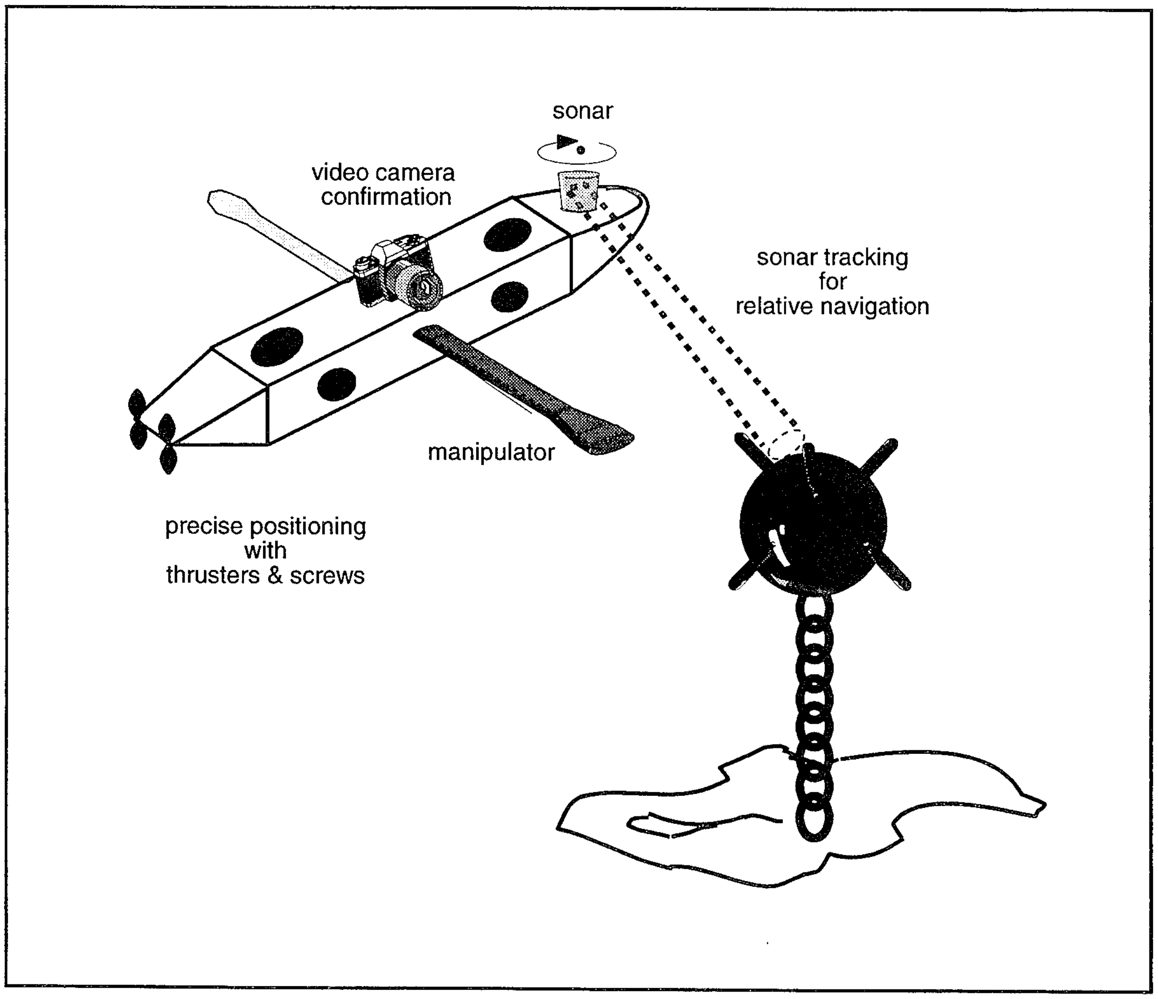

sonar
video camera confirmation
sonar tracking for relative navigation
manipulator
precise positioning with thrusters & screws

Figure 2. Mine hunting concept proposed for Phoenix for localization and neutralization of mines [Brutzman 96].

not be repeated. "Today's approach to mine warfare is the only U.S. naval warfare area in which operating forces routinely bring in the experts to help solve their problem," wrote Chief of Naval Operations Adm. Mike Boorda in a January 1996 white paper on mines. "We can no longer continue to operate in this manner." [Boorda 96]

2. Multipurpose Architecture

Although Mine Warfare is receiving much attention, the general formulation of the control structure for Phoenix can be used for other missions. During the spring of 1992, a workshop was convened at the Florida Atlantic University (FAU) under the sponsorship of the National Science Foundation to discuss and advance the state of autonomy within the field of underwater vehicle technology [Healey 92]. Among the recommendations was the use of inter-institutional technology demonstrations to evaluate the effectiveness of current research concepts. Three sample AUV mission scenarios were selected, each containing significant challenges and sufficient realism to ensure that even partial task completion would produce valuable experimental results. The three task scenarios selected were search and rescue, pollution source location, and navigation with obstacle avoidance. Each mission provides a realistic basis for the employment of AUVs.

B. PROBLEM DESCRIPTION

Although there are far fewer robots designed to operate underwater than in other environments, there is much diversity in the hardware and software of those robots that do exist. Underwater robot hardware is mostly concerned with watertight integrity, maneuvering and sensing. Underwater robot software is usually preoccupied with real-time hardware control. Implemented higher-level functions are rarely as sophisticated or capable as desired.

1. Making Phoenix Truly Autonomous

According to [Byrnes 93], an Autonomous Vehicle is "a self-contained mobile robot with the capacity to sense a dynamic and unstructured environment, plan an intelligent response to that information, and act in a way that is compatible with the accomplishment of a mission, without human intervention." In [Floyd 91], an AUV is defined as an "unmanned submersible vehicle with onboard systems and subsystems that provide motive power, motion control, navigation, obstacle detection and collision avoidance. To be truly autonomous, the vehicle needs to be able to execute a planned mission by controlling and monitoring the onboard systems without any external input. It can replan its mission in case of internal anomalies, such as subsystem degradation, and it will have the capability of replanning its path to avoid previously unknown obstacles."

To call Phoenix autonomous, it must be capable of doing the above functions with only the systems installed onboard. Phoenix has been doing autonomous functions at a limited level at the NPS CAUVR but with one constraint: it needed to use pool-side computers to run some software [Marco 96]. This fails to meet the strict conditions of autonomy.

Phoenix is controlled by the Rational Behavior Model (RBM) [Byrnes 96], a tri-level architecture for control of AUVs. Only one of the three levels was originally on-board the physical vehicle. A means of placing all three levels of the RBM onboard Phoenix had to be found and implemented as part of the work of this thesis.

2. Making Mission Generation User Friendly

The top level of the RBM is the Strategic level. It is written using Prolog, an artificial intelligence language for predicate logic [Rowe 88]. Prolog is known as a readable language, but readability is still far from a user being able to effectively write this code. If this vehicle is to be used in the fleet, it is unrealistic to expect each ship to train individuals on Prolog.

As part of generating a mission, a data file needs to be generated separate from the phase code. This data file contains the actual parameters used by Phoenix, such as depth and positioning information. Creation of the data file was previously done independent of the mission code and stored as a separate file [Marco 96]. There is no automatic means to either generate these position points or to verify their correctness. Writing a mission for the Strategic level was a complex evolution requiring a person proficient at Prolog and familiar with Phoenix.

C. GOALS FOR THESIS

The goal of this thesis project is to accomplish what has been talked about for years, theorized in many papers, and written about in dozens of theses and dissertations, but never accomplished in totality. The goal is clear: make Phoenix a truly autonomous underwater vehicle that can transit in open ocean, find a target, classify it, obtain its position, and return, all without colliding into unknown objects.

Much work has been accomplished on the Strategic (CO) level and the Execution (Watchstander) level. What has not been elaborated on was the middle, Tactical (OOD) level which communicates with the Strategic and Execution levels and accommodates several processes. A Tactical level needed to be installed onboard Phoenix at the time this thesis work was started.

D. ORGANIZATION

The Rational Behavior Module (RBM) robot architecture was designed to emulate the crew of a manned submarine. It is with this context that this thesis is written. Several analogies will be drawn from the submarine watchstanding structure when discussing the software architecture onboard Phoenix. The basic organization of all software components in this thesis imitates the jobs done by crewmembers in much larger underwater vehicles: manned submarines.

This chapter is devoted to the motivation, problem description, and goals for this project. Chapter II discusses previous work in the area of autonomous vehicles and the NPS AUV project. The theoretical development of this project is discussed in Chapter III. Creation of the Tactical level is explained in Chapter IV. Chapter V covers the topic of mission replanning. The expert system created for the Strategic level is the topic for Chapter VI. Chapter VII focuses on the design and results of experiments. Conclusions and recommendations for future work are presented in Chapter VIII. The appendices contain source code, directions on how to obtain the most current version of code, and how to run the software installed.

II. RELATED WORK

A. INTRODUCTION

Research on AUVs is going on worldwide (Figure 3). Concepts and ideas are exchanged through a multitude of conferences and publications. Many theses, dissertations and publications have been written locally involving Phoenix. The focus of this section is to provide a general overview of the evolution of Phoenix hardware and software systems up to where this thesis began.

The Phoenix AUV is an untethered robot submarine designed for research in adaptive control, mission planning, mission execution, and post-mission data analysis [Healey 90]. Phoenix is designed for research and thesis work in artificial intelligence,

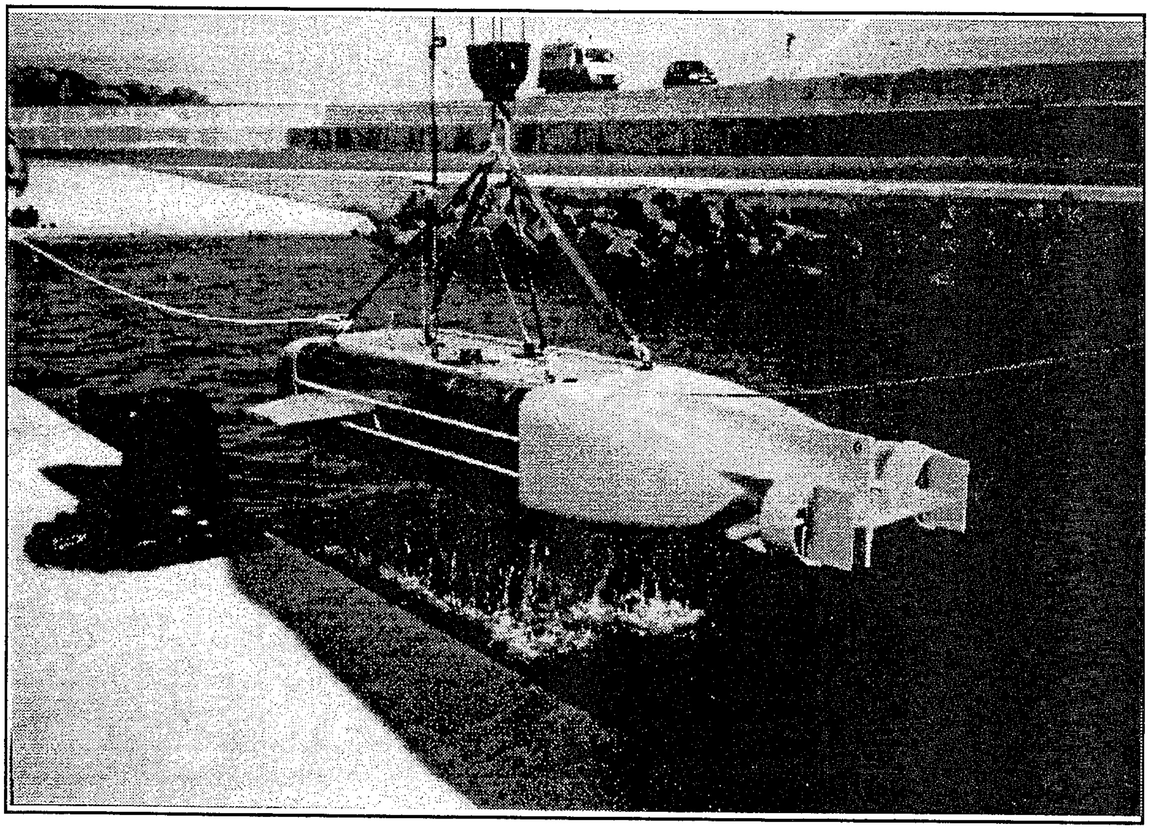


Figure 3. AUV research is going on worldwide. MARIUS is an 18 foot long AUV weighing two tons. It is controlled by 3 GESPAC processors. It is shown here undergoing closed loop lower level testing near Lisbon Portugal in October 1995.

computer visualization and systems integration. AUVs differ from other robots in that they are designed to operate submerged and isolated from communication or external instructions. Missions in the underwater environment require extraordinarily reliable and robust vehicle performance.

B. REAL-TIME PERFORMANCE

To perform many sophisticated mission functions, multiple processes must be operating simultaneously while meeting both hard and soft real-time software scheduling requirements. The autonomous nature of an AUV requires operation without external backup in a harsh and unforgiving environment. Vehicle control, sensor evaluation, underwater navigation, search, path planning, obstacle avoidance, fault tolerance, and numerous other processes are required. All processes must interact with the external environment and each other in real-time with varying degrees of interdependence

It is important to distinguish between hard and soft scheduling criteria for real-time processes. Mission-critical actions such as vehicle control and failure detection present hard real-time scheduling requirements. Failure to meet such strict deadlines may result in mission failure or even catastrophic loss of the vehicle. Conversely, high-level logical processes such as path planning or mission replanning are considered soft requirements, since their execution is rarely mandatory for safe vehicle operation and immediate results are not required. Finally, some processes may have priorities that vary from soft to hard depending on circumstances. For example, path replanning is usually a soft requirement unless a physical object with rapidly closing range rate makes immediate action necessary to avoid collision.

It is important to note that parallelism is equally as important as real-time scheduling for an AUV operating system. This is particularly true if low-level control, complex behaviors, sensor fusion, data analysis, mission planning and many other artificial intelligence aspects of robot mission execution must all coexist and cooperate rapidly.

C. HARDWARE CONFIGURATION

The design and construction of the first NPS AUV began in 1987. NPS AUV I was a two-foot prototype model, with operational screws and gyros, used for the investigation of model-based maneuvering controls, including the automatic identification of significant hydrodynamic characteristics [Healey 90]. The full-scale vehicle NPS AUVII was completed in 1990, after a year of design and construction (Figure 4).

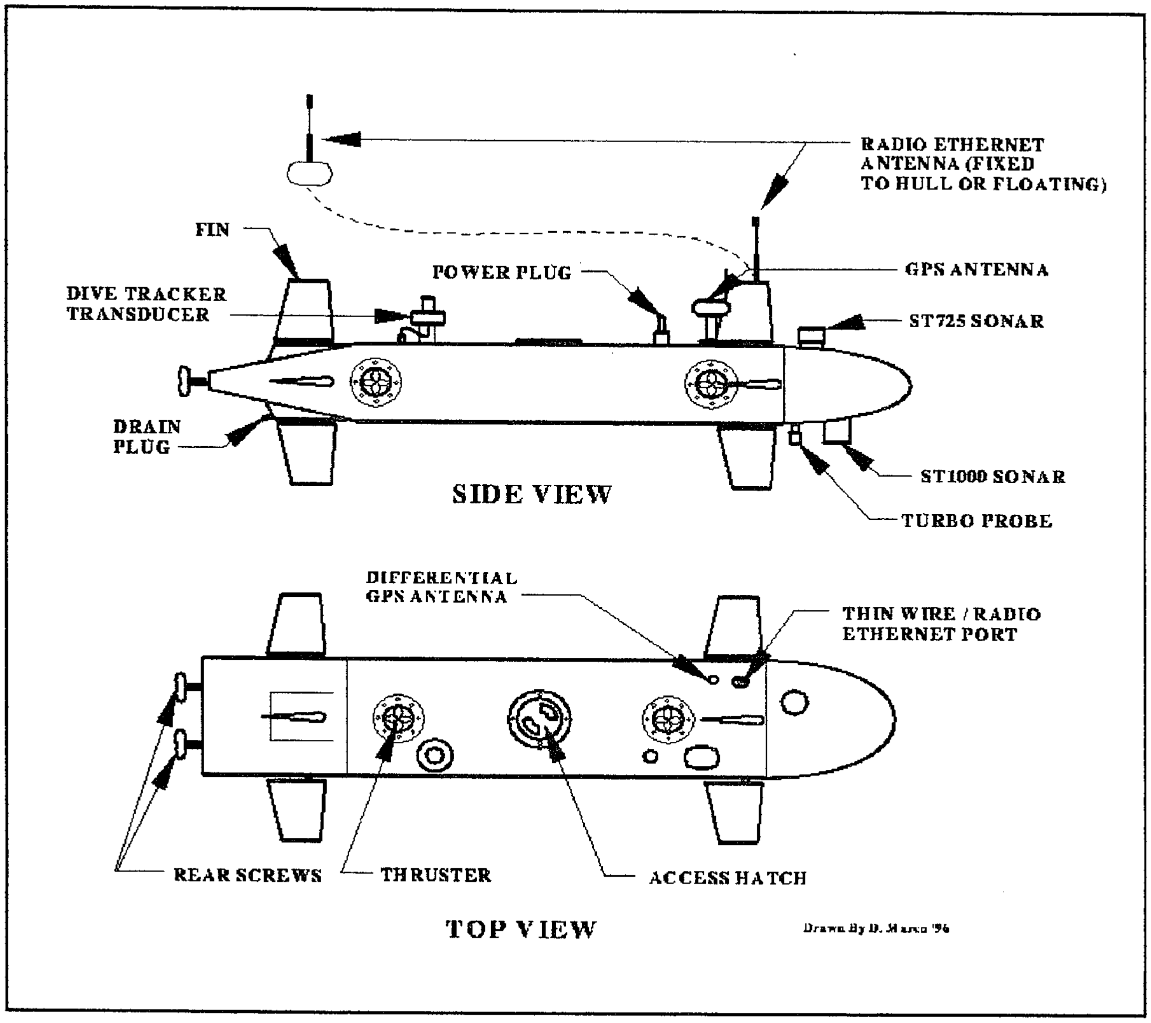


Figure 4. Phoenix AUV external view [Marco 96].

Phoenix is a submersible vehicle capable of a maximum speed of two feet/sec. It has a total length of 93 inches. The main body of the vehicle is made of aluminum and is constructed as a watertight box. It has a beam of 16 inches, a height of 10 inches and a length of 72 inches. The free-flood nose cone is constructed of fiberglass and is 21 inches long. The vehicle displaces 380 pounds and uses fixed ballast.

The vehicle has four forward control surfaces, four aft control surfaces, four tunnel thrusters providing 1/8 horsepower each and two aft screws providing 1/4 horsepower, each powered by 24 volt DC drive motors (Figure 5). With eight control surfaces, the vehicle is highly maneuverable and has a turning radius of three ship lengths.

Phoenix has a depth pressure cell and three sonars. One is a downward-looking sonar, and the other two are mechanically scanned sonars, rotating in a horizontal plane. Two leak detectors are installed forward and aft. A turbine flow-meter probe for water speed measurement, is located beneath the nose cone free-flood volume. Three rate gyros are mounted in various locations to provide roll, pitch and yaw rates. There is also a heading gyro and a vertical gyro.

The vehicle's power sources are lead-acid gel batteries capable of providing up to two hours of propulsion and computational power. Protection from hydrogen gas buildup and explosive hazard is provided by hydrogen absorbers located throughout the vehicle. The Execution level computer is a GESPAC MPU 30Mhz processor with a Motorola 68030 CPU. In addition, the system also has two megabytes of RAM and a 68882 math coprocessor running at 25 MHZ. Serial ports, parallel ports, analog/digital interface cards, and an Ethernet interface are available for internal and external connections. The operating system is the OS-9 multitasking operating system. OS-9 is Microware System Corporation's real-time operating system used by Phoenix for the real-time control of the Execution level. OS-9 is designed to run exclusively on the Motorola 68020/68030/68040 microprocessor family.

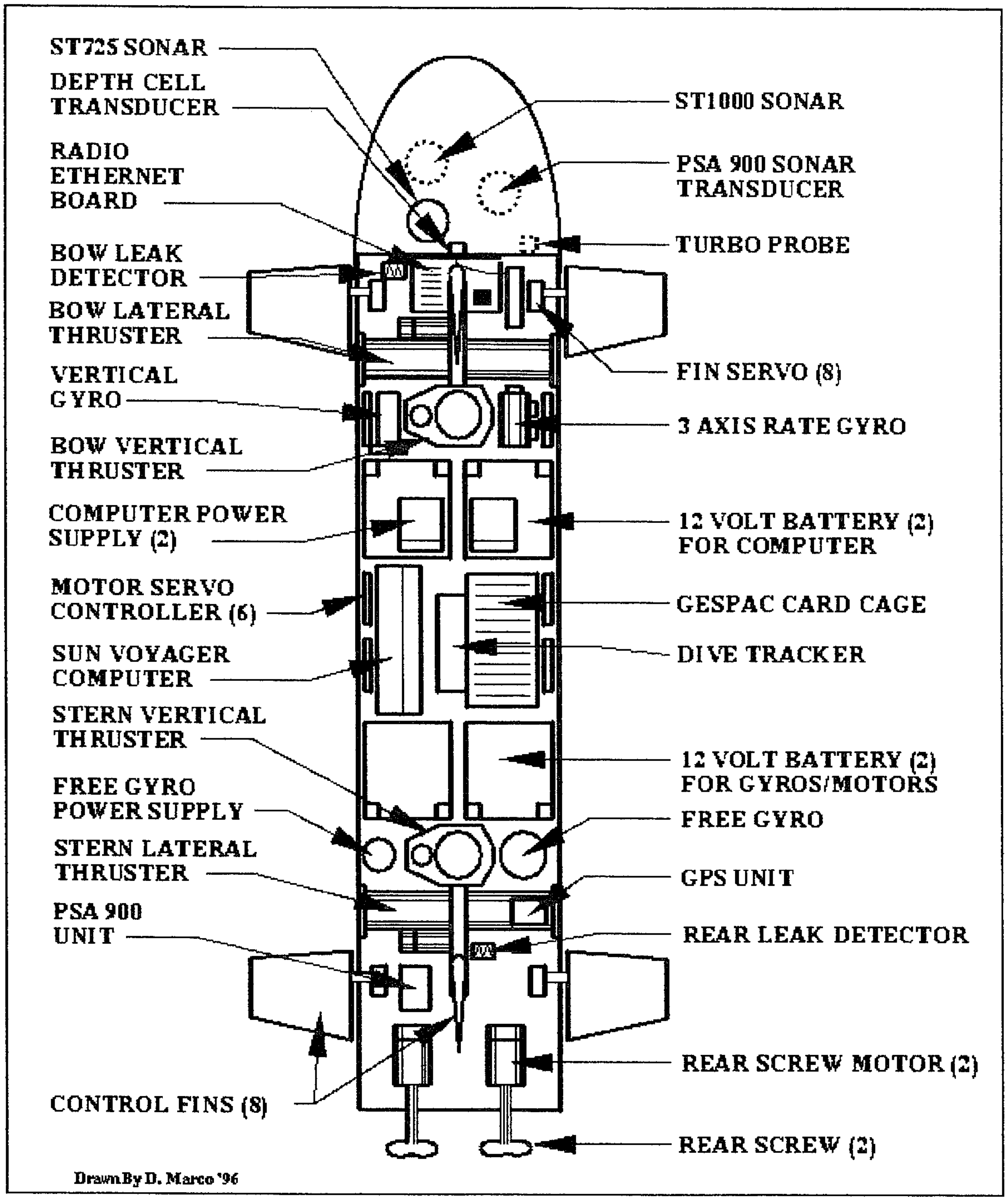


Figure 5. Phoenix internal view. Sun Voyager was added to run the Strategic and Tactical levels [Marco 96].

D. RATIONAL BEHAVIOR MODEL

The software philosophy of Phoenix has been developed extensively since 1987. The result of this development has been the formulation of the Rational Behavior Model (RBM) [Byrnes 93]. This section draws extensively from [Byrnes 93] in describing RBM. RBM is a tri-level software architecture for the control of Autonomous Vehicles. Traditional approaches to the development of software for autonomous vehicle control systems typically involve many individuals working independently, but communicating frequently, to insure consistent interfaces between software components. If a physical system is involved, such as an autonomous vehicle, specialists must be consulted to insure proper integration of the software with the hardware. Construction of a software architecture for the overall control of an autonomous vehicle must involve the coordination of individuals from many academic backgrounds. For example, software using concepts at a high level of abstraction (mission planning and control) must communicate with software concerned with a much lower level of vehicle control (stability and hardware manipulation).

RBM uses the principle of abstraction to simplify the problem of mission control for an autonomous vehicle. RBM uses several complementary abstraction mechanisms in each of its levels [Byrnes 96]. The three levels of RBM, from highest to lowest degree of abstraction, are called Strategic, Tactical, and Execution levels respectively (Figure 6).

The root goal, typically as an overall mission objective, is decomposed within the Strategic level. Resultant higher-order goals of the Strategic level then activate behaviors contained within the Tactical level. These behaviors are designed to produce the actions required by the higher-order goals, in part by generating the commands necessary for the proper operation of the servo loops located in the Execution level. Finally, at the Execution level, the servo controllers are directed by the servo loops to manipulate hardware that cause changes in the relationship between the physical vehicle and its external environment. Various sensors collect data to be ultimately used by the deliberative process contained in the Strategic level which guides future actions.

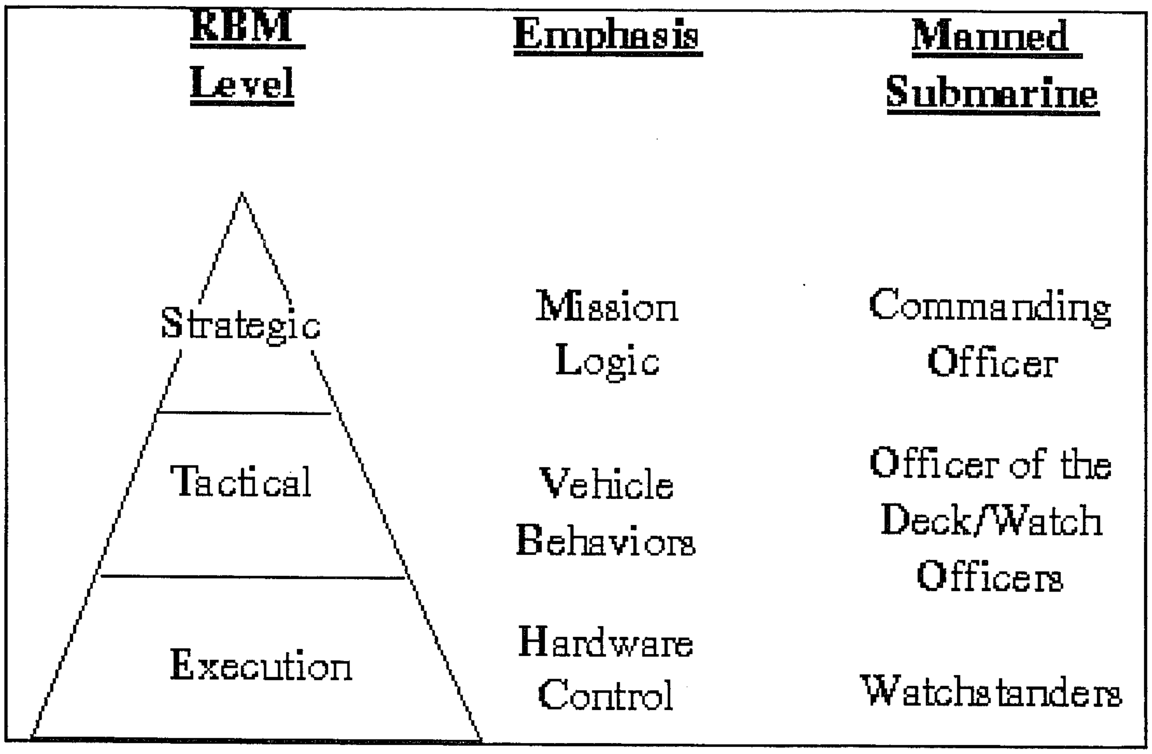


Figure 6. RBM tri-level architecture hierarchy with level emphasis and submarine equivalents listed [Holden 95].

Ultimate control of the vehicle resides with the Strategic level. The search of the Strategic level rule set results in a call to the Tactical level for the initiation of a behavior. Each call is an instance of one of two types: a query or a command. A query is a request for information. Because of restrictions on the use of state variables at the Strategic level [Byrnes 96], replies to these queries must be in binary form. Commands are directives that expect no response other than an acknowledgment that the command has been received. Responses to the decision-making process are obtained through polling using queries. The Strategic level is non-interruptible from within the RBM architecture. Instead, a thread of reasoning, realized through rule chaining, must be allowed to go on to completion. The Tactical level cannot send commands to the Strategic level. Changes in mission state and vehicle environment are reported only when requested by the Strategic level.

Certain circumstances warrant immediate attention, however. Collision avoidance and flooding are examples of conditions which must be responded to outside the normal query-decision-command cycle. These emergencies are best adapted to through the use of reflexive behaviors. This class of behaviors is designed to override existing control to avert problems affecting the safety of the vehicle. The Execution level, operating synchronously and responsible for the direct manipulation of the vehicle's transport mechanisms, is best suited to carry out these casualty procedures.

The Mission Specification is the part of the Strategic level containing the rule set embodying mission specific knowledge. This is the only section changed to run various missions. The Doctrine is the part of the Strategic level containing the mission-independent rule set. It contains the logic common to all missions. Generally doctrine is vehicle independent.

The middle level of RBM, the Tactical level, acts as intermediary between an AI knowledge-based mechanism and the lowest-level vehicle-control subsystem. The primary purpose of the Tactical level is to provide asynchronous coordination between the symbolic-based, behavior-enabling goals of the Strategic level and the numeric-based servo loops of the Execution level. To accomplish this coordination, the Tactical level carries out a finite set of behaviors. The result of a behavior may be a change to the internal state of the Tactical level, receipt and analysis of sensory data passed from the Execution level, a nonroutine data request, or the transmission of commands as numerical setpoints and modes as required for the proper operation of the Execution level subsystems. In addition, upon completion of a behavior, the Tactical level must report completion to the Strategic level. This response is explicit, as a boolean response to a Strategic level query or command.

The behaviors contained within the middle Tactical level of this model are non-logic based, repetitively executed processes, as compared to the rule-based reasoning of the top level. Behaviors may be accomplished by one or more entities within the Tactical level. Behaviors relating to navigation, system checking, sonar interpretation, and obstacle replanning are appropriate for the Tactical level. To isolate the reasoning process of the Strategic level from the vehicle-dependent Execution level, the Tactical level must provide for the complete monitoring and control of the Execution level while being guided by the higher order goals of the Strategic level.

The Execution level is the best understood of the three levels of RBM. The basis of the Execution level derives from the body of knowledge constituting closed-loop servo control theory that is mature, having developed over many decades. Design and implementation of software at this level is mostly accomplished by control engineers, not computer scientists. This level is often taken for granted by the researchers focusing on the upper, more abstract levels of control. The Execution Level of the RBM contains all software entities required to meet hard real-time deadlines and is responsible for basic vehicle stability and safety. Being at the bottom of the RBM hierarchy, the Execution level must provide the interface between the software architecture and the physical hardware of the underlying vehicle. This interface is both an analog or digital signal to control surfaces, motors, payload, sensors, and other devices, and also discrete readings from analog sensing devices such as sonars, pressure gages and accelerometers. Phoenix contains a wide array of hardware that needs to be controlled effectively to produce a vehicle that is stable and able to complete its mission. Doing all of these things concurrently and rapidly are the primary responsibilities of the Execution level.

E. FROM VIRTUAL WORLD TO REALITY

Autonomous underwater robot design is difficult. Unlike most other mobile robots, underwater robots must operate unattended and uncontrolled in a remote and unforgiving environment for prolonged periods. Inaccessibility during operation greatly complicates the design and evaluation of system software. To ensure complete reliability, however, robot software and hardware need to be fully tested in a controlled environment before operational deployment. Such comprehensive testing requirements cannot be met using a standalone laboratory robot due to the complexity and unpredictability of interactions that can occur in the actual remote environment. A different approach is needed which can effectively support research on the many problems facing underwater robot designers [Brutzman 92]. This section draws heavily from information contained in [Brutzman 95]. As reported in [Brutzman 94], an important feature of the Phoenix virtual world simulation is the inclusion of actual vehicle hardware in the simulation. In particular, it was vital to the success of this thesis work that a copy of the GESPAC Execution level computer was included in the virtual world simulation. This feature allowed laboratory testing of Execution level code without the expense and difficulty of premature in-water testing with the physical Phoenix vehicle.

Virtual world systems provide the capability to see and interact with distant, expensive, hazardous or nonexistent three-dimensional environments (Figure 7). A virtual world can provide adequate simulation scope and interaction capability to overcome the inherent design handicaps imposed when building a remote robot to operate in a hazardous environment. A multitude of interrelated requirements necessitates mastering all aspects of world modeling and robot design to build both an authentic virtual world and a capable autonomous robot. Exact reproduction of real-world behavior nevertheless might be undesirable at times. Initial training or evaluation of a new sensor algorithm is best done using the "perfect" error-free model.

People are visually oriented. Being able to see and control location and time in a moving picture allows us to quickly and intuitively understand data sets of much higher dimensionality than is otherwise possible. Having a virtual world available has been a fundamental prerequisite for successful design and application of working robot software.

F. STRATEGIC AND TACTICAL LEVELS

1. Strategic

Autonomous vehicles, by definition, are designed to operate without human intervention. Since a human is not available to make decisions on the spot, an autonomous vehicle must have some capacity to reason. To impart knowledge to an autonomous vehicle, facts and rules describing the problem domain in which the vehicle

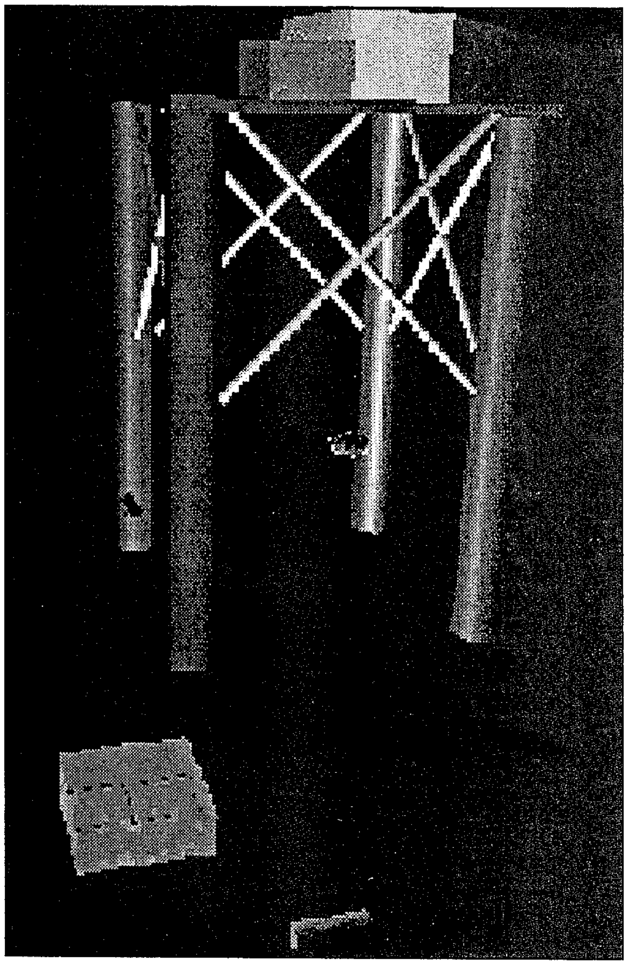


Figure 7. Underwater virtual world provides for rapid code development in a realistic environment [Brutzman 94, 95].

will operate must be placed into a form easily manipulated by a computer. This manipulation is accomplished in an orderly fashion by a reasoning mechanism using a particular control structure. The mechanism is an inference engine, and the control structure is the type of chaining employed by the inference engine.

Prolog is the language being used for the Strategic level onboard Phoenix to provide the reasoning capabilities desired [Rowe 88]. Prolog programs consist of clauses, of which there are three types: facts, rules, and queries. Rules define the problem domain and facts make assertions about the domain known (or assumed) to be true. The set of rules and facts comprise a Prolog database. A query is a statement used to extract information or conclusions from the program. Taken together, the three parts of a logic clause resemble the statement of a mathematical theorem.

The general form of a clause is <head>:-<body>. If the head is omitted, the clause is a query; if the body is omitted, the clause is a fact. A clause with both head and body is a rule. The head and body are composed of relationships, each of which is either an application of a predicate to one or more terms, or an atom. In pure logic programming, because control and logic are separated, the ordering of the clauses is irrelevant to the execution and results of the inference process [Gonzalez 93]. Prolog, in the interest of efficiency and determinism, also includes control mechanisms that greatly influence how programs are written and organized. First, Prolog uses chronological backtracking as a way to explore alternate paths to a goal. If a logical dead end is encountered, the search simply backs up and resumes from the last successful decision point. Second, standard Prolog uses a depth-first strategy when searching through a database. Third, to help reduce the potential drawbacks of these two characteristics, Prolog includes a cut (!) mechanism designed to preempt fruitless backtracking and halt the search for additional solutions when the programmer already knows that none exist.

2. Tactical

A great deal of work has gone into the formulation and creation of the Strategic and Execution levels for Phoenix. What has been least demonstrated until now is the role of the Tactical level. Previously, the main use of the Tactical level was to establish a link between the AI search-based strategic software and the hardware controllers of the Execution level. Message passing between the various levels is not a problem: Prolog code that communicates with C function calls exists, and socket communications between the UNIX operating system and OS-9 system has been established [Marco 96]. The architecture of the Tactical level is discussed in detail in the Rational Behavior Model (RBM) section above. The complete RBM architecture has not been previously implemented onboard Phoenix. Earlier lab versions of reduced Tactical levels are available in [Byrnes 93] [Brutzman 92].

G. SUMMARY

AUV research is an international effort with great diversity in the design of software and hardware architecture. The CAUVR is at the leading edge of this technology, developing and implementing numerous concepts onboard the Phoenix vehicle. Phoenix contains a wide and sophisticated array of hardware and software enabling it to maneuver precisely in the underwater environment. RBM is a tri-level software architecture for the control of autonomous vehicles. Formulation of the RBM has been the result of many years of research on software architecture. It is well developed and is the basis for the current software onboard Phoenix.

In order for a project of this size to maintain a rapid pace of development, a means of developing software quickly needed to be created. A virtual world simulator for Phoenix was developed to provide a means of testing software without the overhead of running the actual vehicle in a harsh environment. As with most projects involving integration of diverse hardware, the software emphasis has been initially placed on interfacing the hardware with control software to best utilize the mechanisms installed. This is true for Phoenix in that the Execution level is well defined. At the start of this thesis research a carefully conceived Strategic level was available and a simple Tactical level that connected the Strategic and Execution level existed. However, full development of the Tactical level had not been achieved prior to the work reported herein. Considered as a whole, many years of painstaking work in numerous fields has been necessary to produce the hardware and software components needed for an effective AUV.

III. THEORETICAL DEVELOPMENT

A. INTRODUCTION

Modeling RBM after the crew of a manned submarine proved to be valuable. The Strategic and Tactical levels developed in this thesis began as a LISP simulation written in part by submariners as a class exercise at NPS. These officers applied their expertise regarding watchstanding principles used in the fleet to the design of the top two layers of control.

Phoenix was missing a key element in order to call it autonomous. Two of the three levels of architecture had to be run by pool side computers due to lack of an onboard computer capable of running these levels. Installation of a computer onboard Phoenix to run the top two levels of software was a key element to the success of this research.

With several individuals working on all three levels of architecture, writing software to accommodate rapid code development was essential. Through the creation of script files and a robust Makefile, software development progressed rapidly.

B. LISP SIMULATION OF STRATEGIC/TACTICAL LEVELS

A simulation of a control structure where the top two levels of RBM existed was written in a course on LISP. The Strategic level was modeled as the Captain/Commanding Officer (CO) of a submarine. The CO issues orders to, and asks questions of, the Tactical level Officer of the Deck (OOD). Action by the OOD determines answers to the questions and commands received by the CO. The only answers the CO accepts are yes sir or no sir! In keeping with the submarine structure, the Tactical level was designed using the precepts governing operational submarine watchsections.

The CO of a submarine issues orders and asks questions of the OOD. The OOD then separates these orders into commands appropriate for the individual watchstanders. If a complex order is received, the OOD will use the Tactical level departments to further

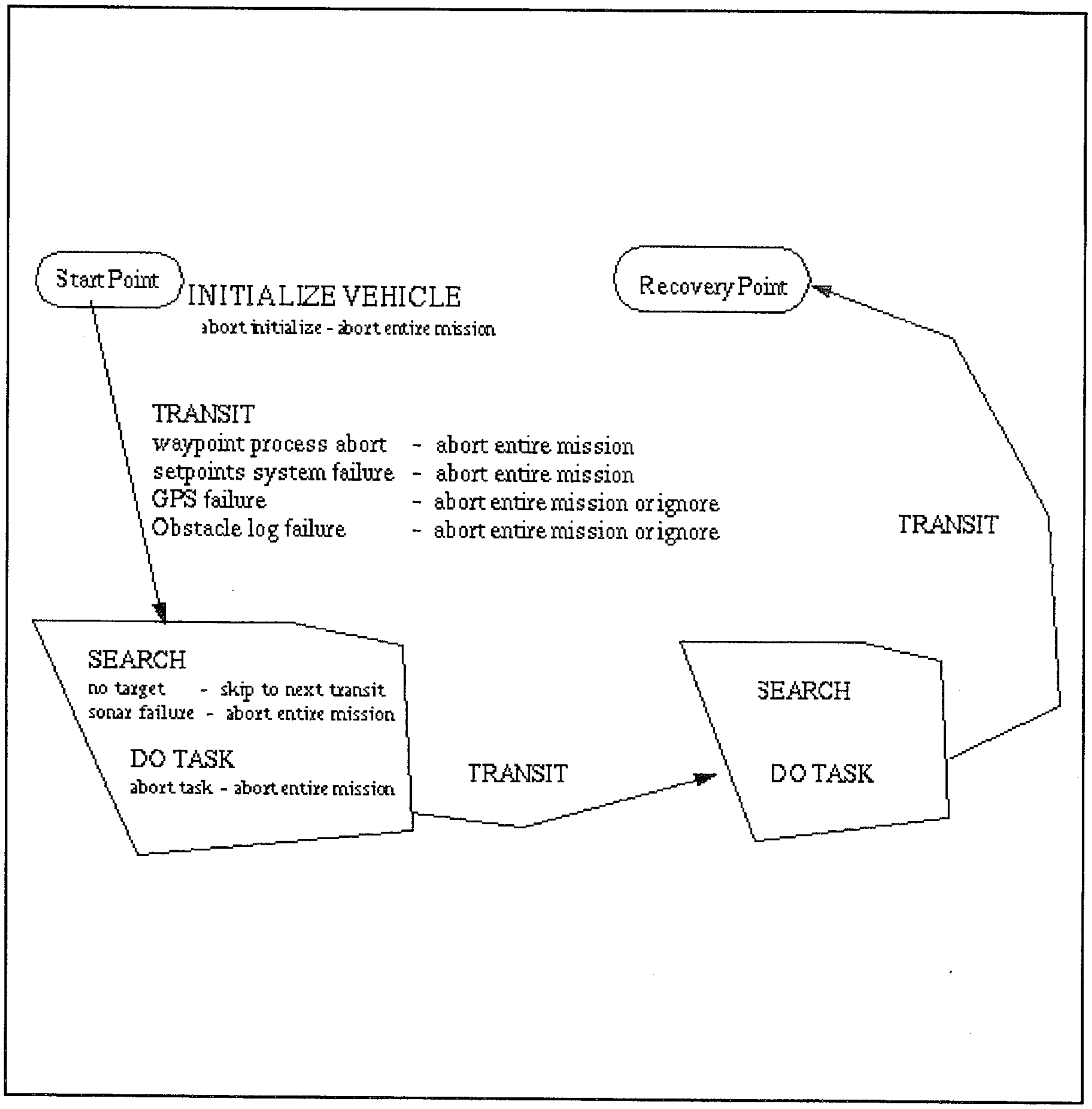

Start Point
INITIALIZE VEHICLE
abort initialize - abort entire mission
Recovery Point
TRANSIT
waypoint process abort - abort entire mission
setpoints system failure - abort entire mission
GPS failure - abort entire mission or ignore
Obstacle log failure - abort entire mission or ignore
TRANSIT
SEARCH
no target - skip to next transit
sonar failure - abort entire mission
DO TASK
abort task - abort entire mission
TRANSIT
SEARCH
DO TASK

Figure 8. A typical AUV mission from the perspective of the Strategic level [Holden 95].

process these commands and provide responses back to the OOD. An Executive officer from a submarine (Michael J. Holden) wrote the CO section (Figure 8), and the author (a submarine OOD) wrote the OOD section. Although RBM was not known by the coders at the time, the independent results from the LISP simulation were very similar to that of RBM.

Given the analogy of a manned submarine, creation of the control structure progressed rapidly. LISP provided an efficient method for testing this control structure. An important concept derived from this simulation is that classes are a useful tool in code development and reduce the amount of code needed. A basic structure was created for communications between the Strategic and Tactical levels. What was still needed was a computer to run it onboard Phoenix.

C. SUN VOYAGER

Why did Phoenix require an external Ethernet connection if it was autonomous? Because insufficient onboard computing power to run the Strategic and Tactical level code, pool side computers were utilized for this purpose. The installation of a second computer for these top two levels of architecture was a critical part of this research. A Sun Voyager was purchased, with 48 Megabytes of RAM and a 3-1/2 inch floppy drive, to fill this need. The CPU is a 60-MHZ microSPARC-II processor and the hard drive size is 850 Megabytes. A twisted-pair Ethernet connection exists on the back of the Voyager along with a SCSI and serial port. The twisted pair connection was connected to Phoenix's thin wire Ethernet via a transceiver that requires a 9-volt power supply. Phoenix has a 12-volt power source that is reduced to 9-volts for the Ethernet transceiver. The total cost of the computer system and accessories was approximately $10,000.

A TRITECH ST725 sonar was to be connected to the Voyager along with the Differential Global Positioning System/Global Positioning System (DGPS/GPS) receiver [McClarin 96]. These sensors both required a serial port connection, while the Voyager had only one. To correct this shortfall, a SCSI to serial converter box was installed on the SCSI connection on the back of the Voyager. With the software provided, this converter box provides for up to four additional serial devices for the Voyager.

With the initial hardware issues solved, attention was turned to the software. The vast majority of the computers in the NPS Computer Science (CS) department run the Sun 4.1.3 operating system. Since the CS administrators were able to support only that system, it was immediately decided to use the Sun 4.1.3 operating system for the

Voyager. The Voyager, however, came bundled with Solaris, a leading 32-bit UNIX operating environment. It was assumed, however, the Voyager would also run the standard Sun 4.1.3. Unfortunately this new computer only accepted the Solaris operating system. This meant that the new computer hardware and new operating system was not going to get any technical support locally.

The Solaris operating system came on a CD and that was all the software received. To run the Strategic level, Prolog would need to be installed. The Tactical level was written in C so that meant obtaining a Solaris-compatible C compiler and installing it. Software compatibility with code that had already been written (before this new operating system, new version of Prolog and new version of C was installed) was a major concern.

A C compiler was purchased and a local copy of Prolog for Solaris was available. Installation of the software went fairly well given the author's unfamiliarity with the system. There were no changes necessary to run the Prolog code. The C code was a little more tricky. With the aid of a few Makefile switches, mostly for socket communication include files, the code that had been written on SGI machines was able to be run under Solaris. Makefile specifics will be discussed later. Another student, David McClarin, ran into many troubles trying to properly set up the serial port on Voyager for his DGPS/GPS unit. He spent several weeks researching and attempting several combinations before obtaining success at correctly configuring and accessing the serial port [McClarin 96].

D. INDEPENDENT LEVELS

Code has been created that runs on SGI, Solaris and O/S-9 machines. Three levels of architecture are written to support RBM onboard Phoenix. The amount of code written and the number of modules created during this project is huge. Many questions remain pertinent. Do the programmers that work on one part of the code need to become an expert on all three levels? As a minimum, does work on the Execution level require the use of the Tactical and Strategic level in the initial debugging phase? Some answers to these questions are provided in the following paragraphs.

One feature designed into the code is the ability to test the Tactical and Execution levels without the need for the next higher level. This capability allowed for work on these two levels to proceed without interaction with the higher levels. This reduced the complexity of coding to the lowest level. In fact, it was not until several months into the project that the new Strategic level program was even written. The Execution level can perform its functions independently of the Tactical or Strategic level. This is accomplished with script files or by using the keyboard. These scripts simulate the orders that would be passed down from the Tactical level. By using the keyboard mode, testing of the Execution level can be accomplished by typing commands one a time to verify the response prior to proceeding to the next command (a useful debugging tool).

The Tactical level can run also using the same scripts as the Execution level to simulate commands from the Strategic level. Also present at the Tactical level is the ability to comment out individual modules to allow for rapid debugging. The result is minimal differences when running the same mission with the simulator on all three versions (Execution with a script, Tactical and Execution with that same script, and Strategic and Tactical and Execution levels running together without a script).

By providing the functionality of scripts, those writing code for the Tactical level did not have to understand or write Prolog code to test their modules. The logical limitation to this feature is that the functionality provided by the upper level(s) is then not available to the lower level(s). For example, if the Execution level is running by itself, there is no Replanning or Fix information available from the Tactical level. For a large portion of the coding process, using the lowest level necessary proved valuable for many reasons. First, it was quicker to test, since less levels needed to be set up and running. Secondly, error localization was easier as less code was involved. Lastly, the upper levels did not need to be created immediately to allow for development and testing of lower level code.

E. MAKEFILE

Creation of independent levels for testing purposes greatly enhanced the rate at which code was produced and debugged. Another tool that quickly became a necessity for code development was the Makefile. A Makefile, at first, was deemed 'nice to have' but its potency became clear when the Tactical level began growing. Compiling the Tactical level code was not a large task initially. But when other programs became integrated, compiling became an enormous task. As the number of files passed 15 (for the Tactical level alone) action needed to be taken to reduce the effort involved in just compiling this set of files. The Makefile for the Tactical level provided the solution to this problem.

One of the key components in the Makefile was to recompile only that file that had been worked on. If code development was occurring on *tactical.c,* waiting the several minutes it took to recompile all files (especially when only one had changed) was tedious. Additionally, if a header file changed, all related files or affected functions needed to be recompiled. This set of recompilation dependencies is provided for in the Makefile, along with a complete recompilation if the Makefile itself has changed. The current Makefile that operates under both Irix and Solaris operating systems appears in Appendix F [Oram 93].

Several different compilation methods can be invoked using this Makefile. Typing 'make clean' caused a complete recompilation of all code. Calling 'make' by itself results in the first compilation in the Makefile to execute. By default this compiles two executable programs: a TACTICAL_STANDALONE version (called tactical) which is used for running the Tactical level without the Strategic level, and a 'strategic' executable for use with the Strategic level Prolog code. Although these are two different executables, the source code files compiled are identical. The only change made is in compilation. A '-DTACTICAL_STANDALONE=0' is included in the strategic version Makefile compilation line. Including the define, TACTICAL_STANDALONE=0 allows different compilations of sections of *tactical.c* based on whether the section is needed for the strategic or tactical version. The major benefit is that only one version of the code needs to be used for both versions of the executable. This topic will be discussed further under the making of *tactical.c*.

F. SUMMARY

Manned submarines have three distinct levels of control to operate and carry out their missions. Emulating this structure for autonomous vehicles proved very useful in the context of the Phoenix project. Once a manned submarine leaves port, it is totally self-sufficient to carry out its mission. Prior to the work of this thesis, Phoenix was not able to leave port due to not having the onboard computing capacity to become self-sufficient. Correcting this deficiency was a major step toward becoming autonomous. Writing software for the new Strategic and Tactical level computer progressed rapidly, in part due to the creation of individualized testing of the lowest levels necessary on the virtual world simulator and the effective use of a Makefile.

IV. THE CREATION OF THE TACTICAL LEVEL

A. INTRODUCTION

The Officer of the Deck (OOD) is the key watchstander onboard a submarine. He is in direct control of the entire submarine while he stands duty. The CO empowers him to faithfully carry out his orders. To accomplish the CO's orders, the OOD has several departments and watchstanders to assist him. The most important part of this thesis research dealt with the creation of the OOD software and structuring a Tactical level that closely resembled that of an operational manned submarine (Figure 9).

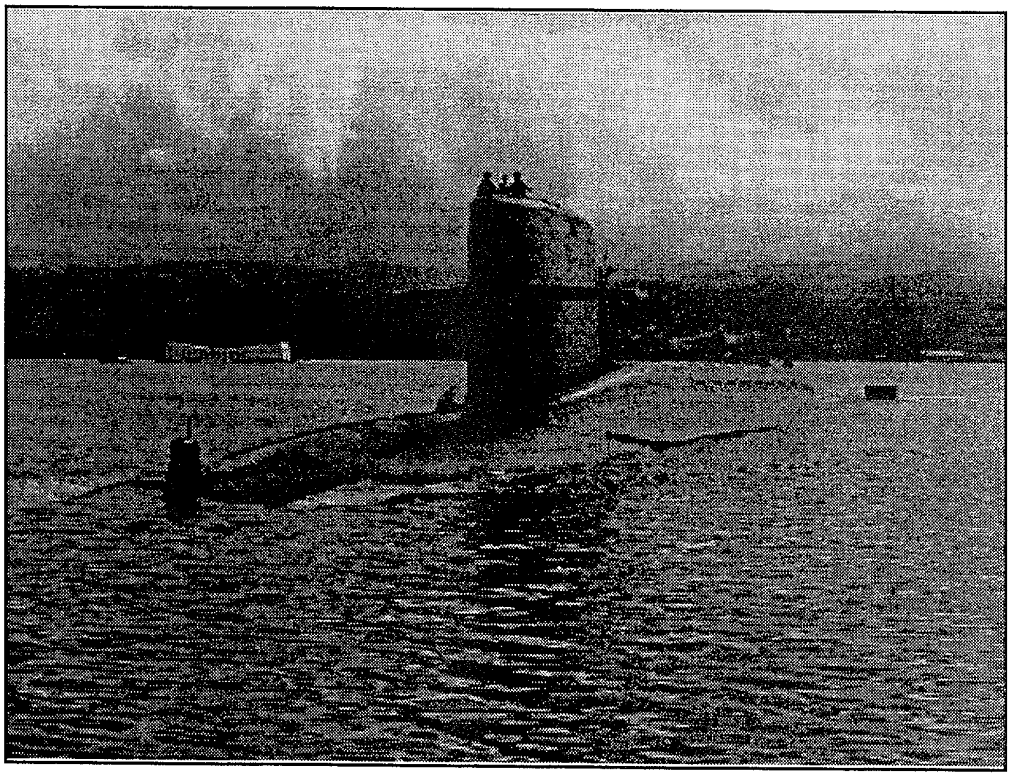


Figure 9. An operational manned submarine.

The Captain (CO) or Commanding Officer, who was created as the Strategic level, speaks to only one individual. That person is the Officer of the Deck (OOD) who is the CO's representative for carrying out his orders during his watch. The OOD decomposes the CO's orders into tasks that are then given to the watchstanders to execute.

The watchstanders (Execution level) perform these tasks and provide the OOD information about the current state of the ship.

B. GENERAL STRUCTURE

The OOD has many responsibilities and cannot allot the time to analyze all of the data coming from the watchstanders. Similarly, the OOD does not have the time to fully decompose all the orders coming down from the CO. To carry out all of the CO's orders while ensuring the safety of the ship, the OOD has several assistants (departments) that work for him at the Tactical level. Each department is specialized in a different tactical area. Some of these departments are continuously processing the data coming from the watchstanders and monitoring the status of the ship to ensure the ship's safety. Other departments are called upon to perform periodic planning or special evolutions.

These departments provide the OOD their expertise, keeping the OOD from getting bogged down unnecessarily with details. This is important so that the OOD can maintain an overall view of the ship's condition by not normally focusing on just one area. The OOD must be constantly looking at the overall operation to ensure the many duties he has been tasked with are being accomplished.

Creation of a Tactical level that will communicate with multiple languages, multiple tasks, and multiple processors is not a trivial matter. What tasks need to reside at the Tactical level? How many processes are needed to accommodate all the functionality required of a "truly autonomous" vehicle? Some of these questions are easily answered as work in those areas has already been accomplished. Other questions require much thought and experimentation to reach a suitable answer.

Most of the tools needed for constructing the Tactical level were already available at the start of this thesis research. Prolog code that communicated with C function calls was already available [Marco 96]. Socket communications that linked the UNIX operating system and the OS-9 operating system were available on the virtual world simulator [Brutzman 94].

Two Execution levels existed for the OS-9 system, one that was currently running onboard Phoenix and one that had only been tested in the virtual world [Brutzman 94]. The Execution level in [Brutzman 94] provided extra functionality not present in the current in-water version on Phoenix and, just as important, allowed the use of the virtual world simulator. However, the robot-specific code that was written for the virtual world was a few years old and never reimplemented on Phoenix. Using the simulator version of the Execution level code would allow testing programs in the simulator and thus the decision was made to upgrade the virtual world code to the current environment of Phoenix. If successful at integrating these two versions, the same code used for the simulator would also be used onboard Phoenix. This task was undertaken by another student [Burns 96]. He was to take the current working in-water version of *execution.c* and integrate it into the virtual world execution code to create a single version.

Because the virtual world execution level contained a large amount of functionality, it provided a rich set of commands and modes to choose from. These commands are similar or identical to those used by human OODs. Since both levels were being worked on simultaneously, close coordination was required between those involved. This was the first time a Tactical level was to be run with the virtual world so changes needed to be made for its use. Once the connection to the virtual world via the Execution level was made, this valuable tool became available to quickly generate and test a large amount of code.

The previous Strategic level was cumbersome and difficult to use. It did not follow the strategy of emulating the CO of a submarine. To create a mission, a data file along with the Prolog mission code needed to be generated. Creating both pieces of code added to the difficulty of mission generation since no automatic checking was involved to verify data matched the phases written in the Prolog code. The Strategic level was performing functions that are better handled at the Tactical and Execution levels. As part of creating a Strategic level Expert System Mission Generator, the Strategic level was rewritten into a smaller, more concise, more readable code including mission data [Davis & Leonhardt 96]. Chapter VI is devoted to discussing that expert system.

The decision as to what kind of mission Phoenix might best accomplish was made several years ago. As previously mentioned, during the spring of 1992 a workshop was convened at Florida Atlantic University (FAU) under the sponsorship of the National Science Foundation, to discuss and advance the state of autonomy within the field of underwater vehicle technology [Healey 92]. Three sample AUV mission scenarios were selected. These were search and rescue, pollution source location, and navigation with obstacle avoidance. Each mission provides a realistic basis for the employment of autonomous underwater vehicles. The second mission was chosen for investigation by the Naval Postgraduate School. This has become known locally as the "Florida Mission" [Byrnes 96].

What departments are needed for Phoenix to accomplish the above mission? How does one create a department in software and how does the OOD speak with these departments? Since Phoenix is an experimental platform, the Tactical level needs to be flexible to adapt to new roles and new hardware. The departments required onboard Phoenix will occasionally change, based on the current mission and configuration of Phoenix. There are however, a few departments that will remain constant with time. They are described in the following sections of this thesis.

1. Navigator

One major capability missing from the current Phoenix configuration was a means of open-ocean navigation. The mathematical models used with the rate gyros and water speed probe are not accurate enough to provide safe navigation for prolonged periods. Two solutions to solve this technology gap were implemented at the same time: satellite navigation and acoustic navigation. DGPS/GPS provides for open ocean navigation by having Phoenix periodically coming close to the surface so the DGPS/GPS unit can obtain positioning data from satellites that can locate the Phoenix within 6 feet of its actual position.

To provide an even more accurate position, although in a restricted area of operation, a short-baseline sonar system known as DiveTracker has been installed (Figure 10). This system was initially designed to keep track of divers from a surface

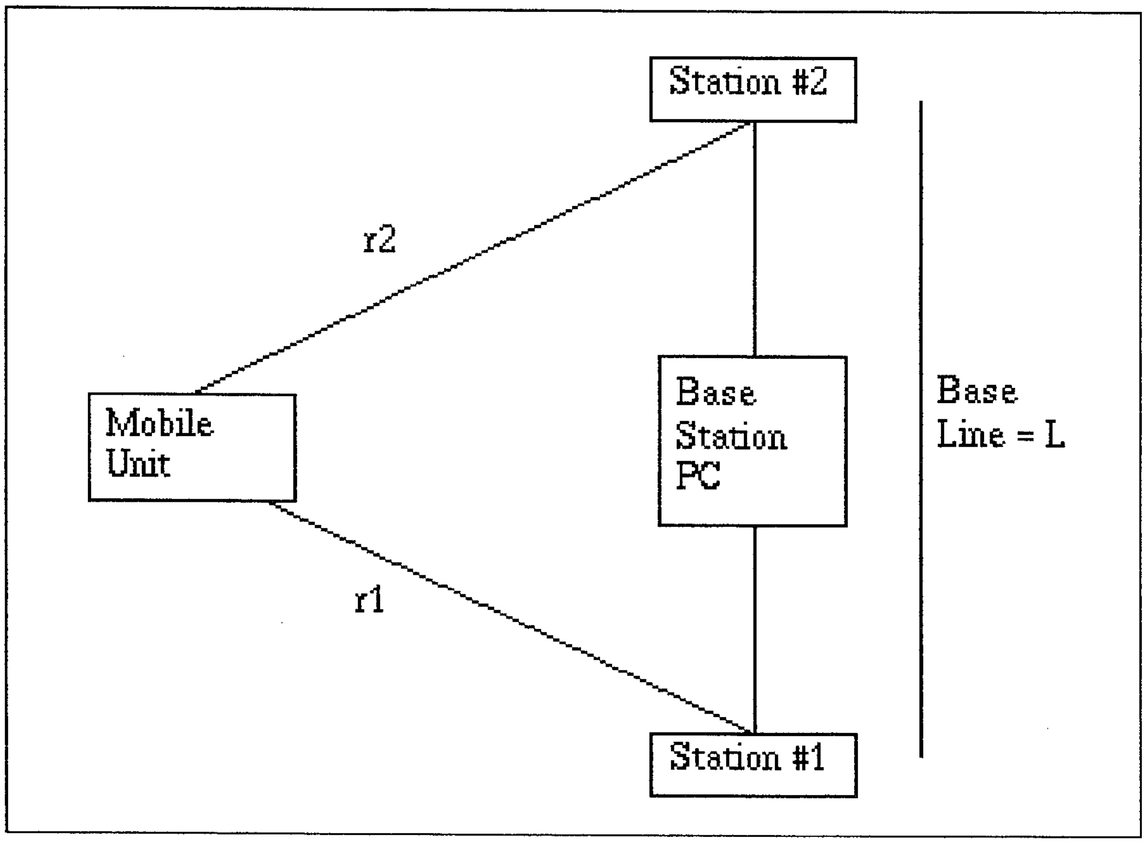


Figure 10. DiveTracker configuration. The mobile unit is mounted on Phoenix [Scrivener 96].

station (such as a stationary boat). It provides precise navigation in local areas where DiveTracker stations can be installed. Phoenix is expected to be able to fix its position within 1 - 2 feet when within a maximum range of several hundred feet from these stations. The DiveTracker was originally installed on the Execution level as there was no Tactical level computer when the DiveTracker was installed. David McClarin and Russell Whalen installed the DGPS/GPS system. McClarin wrote the software to use it in the Navigation department which is a process on the Tactical level [McClarin 96].

The Navigation department is one of the crucial departments. Although the equipment may change, the basic need to accurately determine where the ship is located will always be present. The OOD provides the Navigator with data from the watchstanders. Data obtained from within the Navigation department (DGPS/GPS) and relevant data from the OOD is passed through a filter that determines the best available ship's position [McClarin 96]. This position is sent to the OOD who in turn passes it to the watchstanders so they can reset their dead-reckoned position. As previously discussed, the OOD controls the flow of information, but does not ordinarily get involved with the details of how results are obtained.

2. Sonar

Being able to react with an ever-changing environment is a fundamental requirement in order to call a vehicle autonomous. To react, the vehicle must be able to sense the environment. For an AUV like Phoenix, the major sensors are sonars [Campbell 96]. The sonars are the eyes of the ship. Manned submarines rely heavily on sonar to tell if and where objects are located. Sonar is another one of the key departments that reside at the Tactical level. One of Phoenix's sonars is constantly scanning the water in search of objects (Figure 11). If a new object is encountered, the Sonar department will inform the OOD so that appropriate action can be taken. Sonar will also provide the OOD with a collision threat warning if an object gets too close to the vehicle.

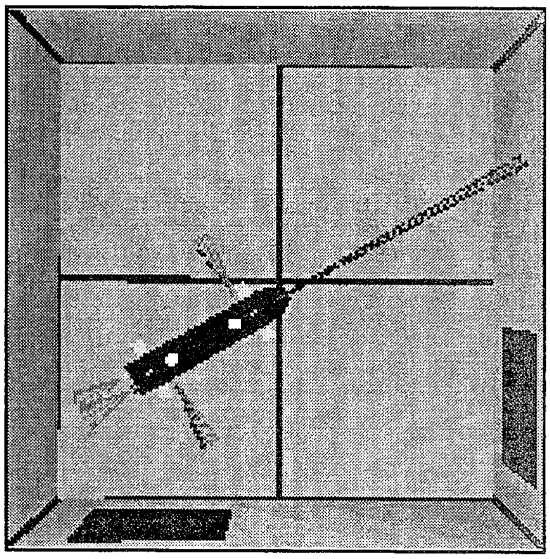


Figure 11. Sonar simulation of Phoenix in virtual CAUVR test tank [Brutzman 94].

3. Replanning

The CO is responsible for the overall mission plan. He decides the track the ship will take. How to maintain the track is the responsibility of the OOD. To determine a safe route to the location the CO has requested, the OOD uses another of his key departments, Replanning.

Each time the CO orders the OOD to change ship's position, the OOD calls upon the Replanning department. The Replanning department is supplied the desired goal location and the ship's current position. The Sonar department (via the OOD) provides the Replanning department with the current environment; (i.e., where all the objects are). The Replanning department takes this information and provides the OOD with the best path to the CO's ordered location ensuring a safety distance around any obstacles. If a new object is found by Sonar while the ship is transiting, the OOD will call upon the Replanning department to check the path that now includes the new object. This department does not constantly process data, but is only called upon when the OOD needs it.

4. Engineering

Monitoring of the environment is essential for an autonomous vehicle. This includes not only the external environment, but also the internal environment of the vehicle. The OOD needs to know if one of the ship's systems is degraded or not functioning, so appropriate corrective actions can be employed. Monitoring of Phoenix's internal environment goes to the Engineering department. The Engineering department is responsible for checking various parameters such as battery voltage, fin movement and screw rpm to decide if they are operating in a degraded mode or not at all (i.e., battery voltage drops below a setpoint). This department is responsible for informing the OOD when an abnormal condition occurs and to provide diagnostics to troubleshoot problems. An example might be if an order was sent for fins to go 20 degrees and data from the Execution level indicated no movement. The Engineer would diagnose this problem and recommend cycling the planes to troubleshoot the problem. If the planes still indicate failure the Engineer would provide the limitations to vehicle operation to the OOD. At this conjuncture the Engineering department has not been created. However, the framework has been created at the Tactical level for this department when it is created. Most of the functionality of an Engineering department is currently carried out by the Execution level.

5. Process Creation

Three departments are currently installed on Phoenix. Each is given its own memory space and time slice of the Voyager CPU. This is done by forking each process (Figure 12). New processes are created in UNIX when an existing process calls the fork() function. Each new process is known as a *child process*. Both the children and parent continue executing with the instruction that follows the call to fork. Each child is a copy of the parent. The child gets a copy of the parent's data space, heap, and stack. The parent and child do not share the same physical portion of memory. To prevent the duplicate execution of the remaining code, a call to a function which is the code for the child process ordinarily occurs within the block that forks the process [Stevens 92].

```
If ((Engineer = fork ()) == 0)
{
    printf("ENGINEER Module forked \n");
    Eng_mod();
}
```

Figure 12. Forking a process [Stevens 92].

Unless otherwise indicated, the code fragments used in this thesis are in located in the file *tactical.c*. A complete listing of *tactical.c* appears in Appendix A. The process ID is stored in the variable Engineer. Eng_mod is called within the brackets following the fork process and therefore will be called only by the Engineer child process. Each child process is set up with the same format (Figure 13). Such a process is forked and then immediately calls a function that only that specific child process uses. Each child

```
/*****************************************************************/
/* Eng_mod ()                                                     */
/*****************************************************************/

{
    TACTICALPARSE = 1;
    for ( ; ; ) /*     loop forever       */
    {
        strcpy (String_read, "");
        strcpy (String_back, "");
        if (read Eng_telemetry_fd[0], String_read, MAXBUFFERSIZE)
            == -1) {}
        else
        {
            parse_telemetry_string (String_read);
        }

        if (read (OOD_to_Eng_fd[0], String_read, MAXBUFFERSIZE)
            == -1) {}
        else
        {
            if (strcmp (String_read, "QUIT") == 0)
            {
                printf ("Terminating Engineer Module \n");
                exit (0);
            }
         }
         write (Eng_to_OOD_fd[1], String_back, MAXBUFFERSIZE)
    }
}
```

Figure 13. Child process function is called immediately after spawning process.

process function call contains an infinite loop that keeps the child process inside its own unique program. Without such a loop the child process might return to the parent process code and commence duplicating the actions of the parent process. Exit (0) terminates the child process and is triggered when the parent process (OOD) sends the string 'QUIT'.

C. INTERPROCESS COMMUNICATIONS

The next question to answer is how does the OOD talk with the departments and vice versa? The OOD needs to send commands and data to the departments and receive responses back from them. Since each department and the OOD are processes on the same processor, the pipe method of interprocess communication (IPC) is used.

1. Pipes

The oldest form of UNIX IPC is a pipe. Pipes are only allowed between processes on a single processor. A pipe is created by calling the pipe function (Figure 14). The OOD_to_Eng_fd[0] file descriptor is opened for reading and the OOD_to_Eng_fd[1] file descriptor is used for writing. The pipes are created before forking the child process. When the child process forks, it creates an IPC channel from the parent to the child and vice versa. Using the read and write function calls with the pipes allows communication between the parent and child processes.

```
#include <unistd.h>
int OOD_to_Eng_fd[2];
int pipe (OOD_to_Eng_fd);
```

Figure 14. Pipe creation.

2. Telemetry Pipe

Two incoming pipes and one outgoing pipe are created for each department. The reason for two separate pipes from the OOD is due to the two types of information sent on these pipes. Eng_telemetry_string _fd[0] is used for passing the telemetry string that comes from the Execution level to each department.

The telemetry string contains all the state vector data sent from the Execution level. Items such as position, speed, DiveTracker ranges etc., are placed on this string. In keeping with the philosophy of letting everyone know everything, the entire data string is sent to each department. Although the variables relating to this data are global, the local copies of values are not updated automatically in each process. The forking process creates a new memory location for these variables and the current values at the time of the forking process are put into them. After the forking process, the only means of updating these variables is within the processes. Telemetry strings thus need to be sent to each department every time the state vector is updated.

Functions outside the child process are still available for child process use. As seen above the function call parse_telemetry_string (String_read) exists in the Execution level code. Since each process needs to break down the strings it receives (String_read) into its individual components: x, y, z, phi, theta, psi, etc., it would have been incredibly redundant to write a separate parsing function for each process. By including the appropriate header file (Figure 15) the parse_telemetry_string function becomes available

```
#include "../execution/statevector.h"
```

Figure 15. Statevector definitions include file.

for all processes to use. This allows for maximum code reuse and thus reduces the size of programs. The telemetry string is updated by the Execution level each time it goes through its cycle.

Only the most current telemetry string is required by each department. The OOD performs a read on the state vector telemetry file to clear any outdated data immediately prior to writing the new telemetry string to that pipe (Figure 16). This ensures that if the child process has not yet read the pipe, only the most current data is available. If the

```
read (Eng_telemetry_fd[0], Read_to_clear, MAXBUFFERSIZE);
```

Figure 16. Clearing the buffer prior to writing.

pipe was not cleared, it is be possible that the pipe buffer might fill and block the parent process, or (even worse) the department would be acting on outdated data. This can occur if the child process is not looping as fast as the parent process.

When a process forks it is allowed a certain amount of CPU time. Each process executes within an infinite loop and therefore may loop several times during its allotted CPU time. To prevent the child process from blocking on a read from the parent, a nonblocking read was established. Another pipe was created to send commands and other non-time-related information reliably. The telemetry pipe can not be used as it gets cleared each time through, making it possible for such commands to be erased before the department has a chance to see them.

The (OOD_to_Eng_fd[]) pipe does not perform a read_to_clear which allows the commands to build up on the pipe if the child process is running more slowly. It is important to remember that the traffic on this type of pipe must be limited so that a large amount of data does not get put into the pipe's buffer causing a block to occur. Nor can the pipe be cleared each loop or else those commands sent on that pipe can be lost. The only remaining issue between the OOD and the departments is the common vocabulary that they are to use when communicating with each other.

Each department parses the string sent to it by the OOD. It then examines the elements of the parsed string to decide what, if any, actions are required. Inside each department a string compare occurs. An example is the QUIT command sent by the OOD (Figure 17). Had the OOD sent the command STOP, the string compare would have failed and the process would continue running. During the development of the Voyager code, early in the coding process, each programmer writing a department discussed the inputs needed by the department and the results these departments were providing. Once these functional specification parameters were established, programmers

```
if (strcmp (String_read,"QUIT") == 0)
{
     printf("Terminating Engineer Module \n");
     exit(0);
}
```

Figure 17. Terminating a child process using exit(0).

were able to code individually. The OOD would supply them with their requested information and would also create appropriate actions based on the agreed upon outputs. How they go about determining these outputs is irrelevant to the OOD. This modularity of design allows for parallel programming to occur and simplifies the addition of further software modules (i.e., departments).

As discussed previously, several departments have been created for Phoenix to assist the OOD in its duties. A Navigation department was created to generate the ship's position. A Sonar department was implemented to be the "eyes" of the ship. To ensure safe travel around obstacles the Replanning department was written. A valuable (but not yet installed) Engineering department is needed to monitor ship's functions. The creation of these departments starts with the forking of their process and a call to their function. These functions contain an infinite loop and are terminated via the exit(0) function. In order for these departments to talk effectively with the OOD, several pipes have been created. Those with time-sensitive data are cleared before each write. The reads are set to nonblocking so that the process can continue even when new data is not available.

D. CONNECTIONS TO STRATEGIC AND EXECUTION LEVELS

The communications between the OOD and its departments are via IPC. How does the OOD talk with the CO or his watchstanders (Execution level), which may reside on different processors? How can Prolog communicate with a C program? How does the Voyager computer with Solaris 2.4 operating system communicate with a GESPAC computer running the OS-9 operating system since pipes will not work outside of a single processor? These questions are the topics of the following paragraphs.

1. Compiling with Prolog

Quintus is one of the major producers of Prolog compilers [Quintus 91]. They have included many utilities that are not available in other Prolog versions. One of the more significant features of Quintus Prolog is that of foreign function calls. Otherwise known as "hooks to C," these function calls provide the mechanism for communication between C and Prolog code. Foreign functions are loaded directly into the Prolog system by using one of the built-in predicates, load_foreign_executable/1 or load_foreign_files/2. These predicates load executable images or object files into the address space of the running Prolog. When load_foreign_files/2 is called, it calls the hook predicates foreign_file/2 and foreign/3 in the current source module.

During the redesign of the Strategic level, an effort was made to make it as simple and as readable as possible. To that extent only one foreign_file and one foreign_function is used (Figure 18). This is a great reduction from the previous Strategic level code. The file *tactical.o* is then loaded. This file is not the file created by *tactical.c*, but rather it is the combination of all the object files (*.o) related to and including the *tactical.c* code, that also consists of some object files from the Execution level. The only function called from the tactical.o code is the ood (cmd) function. The +string above indicates that Prolog is sending the function a string. The [-integer] indicates that an integer is returned from the function call.

```
foreign_file (tactical, [ood]).

foreign (ood,c,ood(+string, [-integer])).
```

Figure 18. Quintus hooks to C code functions.

2. Socket Communications with Execution Level

The GESPAC computer system provides real-time control of the hardware components at the Execution level. It is recognized that the OS-9 system is outdated and needs to be replaced. However, to accomplish this will require a major rework of the Execution level device drivers. Instead, the Execution level, for the purposes of this thesis, will be considered a black box. Commands are sent to this black box and it ensures that they are carried out. How the digital to analog conversions occur and what control algorithms are used to maintain depth do not need to be known by the OOD. This follows the modularity concept of only needing to know the inputs and outputs of system modules.

There is one widely available technique for communications across processors: Berkeley Standard Distribution (BSD) sockets compatible with the Internet Protocol (IP) [Stevens 92]. IP-compatible socket communications are implemented on all computer platforms, and are available as auxiliary function libraries in most programming languages of interest. Use of sockets has several added benefits: processes can run independently, interchangeably, and remotely on vehicle processors or networked workstations.

Phoenix has a thin-wire Ethernet network interface installed. The Ethernet cable connects the two onboard computers and has a through-hull connection to optionally allow for external computer connectivity. The Center for AUV Research (CAUVR) lab is wired in thin-wire Ethernet and can be connected to Phoenix for pool side testing. The lab is connected to the Internet via a wireless Ethernet bridge to the NPS campus several miles away. Although all processes are running in the Phoenix processors, by use of a telnet window to the onboard processors these processes can be monitored and interrupted if necessary. This allows for interaction with the vehicle while in the testing and developing phase of building the code.

E. MAJOR FUNCTIONS IN *TACTICAL.C*

This thesis has answered the questions of how to create a Tactical level that communicates with multiple languages, multiple processors, and multiple tasks. A description of the tasks required to create a "truly autonomous" vehicle at the Tactical level has been given. The interfaces between the levels are explicitly defined. What has not been discussed yet is how the code accomplishes these jobs. What are the specific duties of the OOD?

A programming style obtained from [Brutzman 94] was incorporated. It entails placing many print statements around key sections of code and placing the "if (TRACE)" switch around them (Figure 19). If a code error occurs and is not readily visible, by setting TRACE = TRUE, all these print statements become activated which then produces a verbose listing during program execution.

```
if(TRACE)

    printf("Commencing Replanning on file %s\n" ,sonar_file_ name);
```

Figure 19. Trace statements produce verbose listings during program execution.

1. Two Different Executables From One File

Creating a Tactical level that would both run by itself (STANDALONE) and with the Strategic level (Prolog version) is quite challenging. For several months two completely separate programs were used, requiring a lot of maintenance, and often leading to what is referred to as "versionitis." Versionitis is a plague that infects one's code due to attempting to maintain more than one copy of the code and switching between them. The result is often not knowing which copy of code does what, and old errors recurring when running what was thought to be the current version.

The combination of these two program copies created challenges in several areas. First the STANDALONE code contained an infinite loop, just like the child processes. The Prolog version does not contain an infinite loop as the Prolog code provides the looping. The Prolog code expects a 0 or 1 (FALSE or TRUE) returned each time it calls the Tactical level while the STANDALONE code did not. The STANDALONE version required a main () function while the Prolog version would not allow for a main (), instead requiring a separate function call, ood (cmd). The ood(cmd) function is passed parameters from the Prolog code while the STANDALONE code read from a script file. Although at first there seemed to be an insurmountable amount of differences, the resulting code to accomplish these changes was eventually completed without excessive effort.

As explained in the Makefile section, the -D TACTICAL_STANDALONE causes a #define to be generated which allows sections of code to be compiled or excluded from compilation. By use of the TACTICALSTANDALONE preprocessor boolean definition, conditionals can be placed around version-specific code. The key was to minimize the sections that are dependent on the version running. To do this, several changes were made. For example the section of code that looks at the command received was moved out of the main () and placed in the function call string_compare.

Once these sections were removed from main (), little code remained which was specific to the TACTICALSTANDALONE version. A #define switch was placed around the main function to compile it when using that version, or to ignore it when using the

Prolog version. The remaining section of main () was reduced to calling the initialize () function and the creation of the infinite loop that merely zeroes two variables and calls the ood_command_loop (). All that was left to do was to add the ood (cmd) function that Prolog was looking for. Thus a single version of the *tactical.c* program is now available that can compile under all configurations. The ood (cmd) function is not called by the TACTICALSTANDALONE version and is therefore only called by the Prolog code. Inside the ood (cmd) function the string received from Prolog is parsed, and the command is saved as strategic_command, and then the ood_command_loop () function is called. Following the call to ood_command_loop () and subsequently the call to string_compare (), the RETURN_VALUE from string_compare is sent back to the Prolog code or ignored by the TACTICALSTANDALONE version. The result is all the code is used by both versions except for approximately 10 lines of code (Figure 20). There is no further need to maintain two separate versions of code. This represents a major productivity and reliability gain.

2. Scripts

To test the effects the Tactical level has on Phoenix when advancing from the Execution level running alone with a script to the Tactical level running with the Execution level, it was decided to make the Tactical level capable of running from scripts. In fact the Tactical level calls the same script file from the Execution level when it is running in TACTICALSTANDALONE mode. This powerful technique required careful coordination between the Tactical level code and the Execution level code. One of the major changes required in the Execution level code was that the Tactical level now controls when the next command was to be read from the mission file. The Execution level had to inform the Tactical level when conditions were satisfied; i.e., when a hoverpoint was reached. The decision was made to have the Execution level send a STABLE (command) message when these conditions were met. This message triggers an appropriate response in the Tactical level that the most recent command was achieved. These conditions might have been checked at the Tactical level, but since they were already designed into the Execution level code it was decided to maximize code reuse and

```
/*******************************************************************/
/*  main () for use with TACTICAL_STANDALONE (No Prolog)           */
/*******************************************************************/

#if defined(TACTICAL_STANDALONE)

main()
{
    TACTICALSTANDALONE = TRUE;
    initialize();
    printf("Initialization complete \n");

    for ( ; ; )
    {
        strcpy(command_sent, "");
        strcpy(command,"");
        ood_command_loop ();
    }
}

#endif

/*******************************************************************/
/*        ood(cmd)  for use with the Prolog code                   */
/*******************************************************************/

int ood(cmd)

char * cmd;

{

    parse_command(cmd);
    strcpy (strategic_command, command);

    ood_command_loop ();
    return (RETURN_VALUE);
}
```

Figure 20. The only version-specific code that needs to be separated to run the two different executable programs. A precompilation switch is used to either add or ignore the main () function.

not perform a redundant task at the Tactical level. Single versions of source code for the determination of achieving commands at the various levels eliminates versionitis - at least in this case. Tactical and Execution levels are thus guaranteed to always speak and understand the same language.

There are several key functions used by the Tactical level to accomplish its mission. Some have been discussed briefly: string_compare, ood_command_loop, and others. The rest of the major functions and programming styles used will now be discussed. Some boolean switches and function names used have long names. TACTICALSTANDALONE is one of these switches. The intent is to make function and variable names meaningful. With code that runs into the thousands of lines, it is imperative that the code is easily understandable, not only by the original programmer but by the many individuals who interact with that code at present and in the future.

3. Initialization

The initialize () function is one of the first functions called in the tactical.c code. It sets up the socket communications with the Execution level, creates the pipes, and forks the departments. It opens the mission.script file if TACTICALSTANDALONE is TRUE. Otherwise, it opens the initialization.script file for the Prolog version. The initialization.script file (used by the Prolog version) contains the parameters needed by the departments and Execution level to properly initialize the vehicle. These parameters include the position of the DiveTracker stations, the initial position of Phoenix and the gyro error, as a minimum. Once these parameters are read, a software timer starts that gives Phoenix a short period of time to complete the initialization process.

Each department and the Execution level informs the OOD when they have completed their initialization process. Once all departments and the Execution level reports they are initialized, the OOD will inform the CO the ship is initialized. From there, the CO commences with the mission orders. If the ship fails to initialize, the OOD will inform the CO and the mission will likely be aborted. Since the CO only accepts Boolean answers, it is the CO's responsibility to query the OOD for appropriate responses. In the case of initializing the vehicle, the CO commands the OOD to initialize. The command is followed by a start_timer call with a time. The CO queries the OOD by asking if initialization is complete. If the processes are still initializing, the OOD responds negatively to the CO by returning a 0 (FALSE). The CO will then ask if a time out has occurred. The OOD will check the time elapsed since given the start timer order and if the current time exceeds the time out, a positive response is made to the CO by returning a 1 (TRUE) to the Prolog code. The positive response to the time out query for initialization will trigger an abort mission from the CO.

4. Processing Pipe Streams

Process_pipe_streams () is called to evaluate the strings sent to the OOD by the departments. Each time the OOD goes through its loop, it reads and processes the strings from the departments. With the information received from the departments, the OOD will determine actions to take and either call other departments for assistance, send executable commands to the Execution level, or respond to the CO.

5. String Compares

String_compare () is a function that does what its name implies. It uses the strcmp () function to check the strategic_command received (either from the mission script or from Prolog) against a series of expected commands (Figure 21). When the

```
int string_compare ()
```

Figure 21. String compares used to evaluate which command was sent.

match is found, that block of code is executed. At the top of each block the boolean flags and variables are set. Command_sent is the global variable used by write_to_execution_level_socket () to send a string to the Execution level. At the end of each block is the RETURN_VALUE. The reason the return () function is not used in each block is that there is additional code following the string_compare () function that needs to be executed.

6. Parse Commands

In the OOD code, variable1 is used. The value for variable1 comes from another major function, parse_command (). Parse_command is a multipurpose function that parses a string into its constituent parts. Each string has a command keyword as its first parameter. Parse_command does some evaluation of the keyword and performs actions based on the command. Boolean flags are set, file writing occurs, variables are set and functions are called based on the command received. One of the lengthier parts of the parse_command () function deals with the command STABLE. The STABLE command, as discussed earlier, comes from the Execution level. It is used to indicate when a waypoint, hoverpoint, or GPS fix is completed. The handling of SUBWAYPOINTS also occurs within this section. The commands that come from the Tactical level departments are handled in a separate but similar function: process_pipe_streams().

F. SUMMARY

Of all three levels of architecture of the RBM, at the time this thesis research was started the Tactical level was the least developed onboard Phoenix. Creation of an OOD module as the focal point in the development of the Tactical level is one of the key concepts pursued in this thesis. The OOD controls the flow of information between three levels and within the Tactical level, yet it does not get bogged down in unnecessary details. Instead, by forking processes, the OOD makes several departments available to assist it in the processing of commands and data. Code reuse by including appropriate header files reduces the amount of code needed to be written for the Tactical level. By use of a modular design, only the interfaces between sections of code needed to be described before parallel programming could occur. In this way the departments and OOD were all developed simultaneously.

V. PATH PLANNING

A. INTRODUCTION

Phoenix is progressing rapidly as an autonomous vehicle. A major part of being truly autonomous is being interactive with the external environment. An important part of this interaction is the ability to avoid collisions with objects that were not originally planned for (Figure 22).

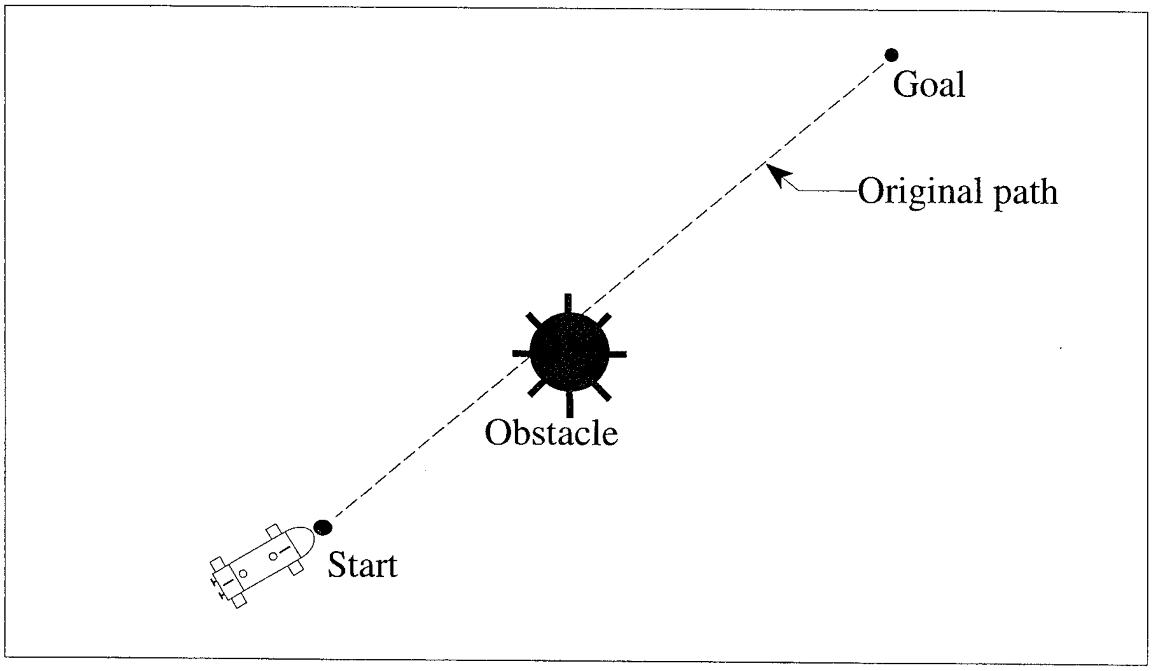


Figure 22. Initial straight line path will cause a collision with the obstacle.

A big step in the direction of achieving obstacle avoidance in Phoenix came in the creation of the NPS AUV INTEGRATED SIMULATOR [Brutzman 1992]. This work created code, in C, to accomplish an A-star (A*) or Dijkstra shortest path finding for a circle world environment and also provided the foundation for the Replanning department onboard Phoenix. Robot path planning is the process of finding an allowable, safe or optimal path for a robot to travel between locations. In path planning, the decision of which model to use to represent obstacles that a robot must avoid is critical. The following is a brief summary of this work.

B. SHORTEST PATH

Path planning's objective is to find the shortest path or a relatively short path from a starting point to a goal point, avoiding all known obstacles in the area. Such a search results in a preplanned mission that is then stored in the mission code. Since Phoenix is operating in a real environment, not all obstacles will be accounted for in the mission planning stage. This results in the need for the vehicle to provide a replanning capability.

A well-known and fundamental method used for path planning is exemplified by the configuration space approach [Lozano Pérez 79], summarized in [Brutzman 92]. In the configuration space (c-space) approach, a world is modeled as a set of geometric obstacles. Obstacle boundaries are expanded to include the effective radius of a mobile robot. The robot center is then treated as a reference point. The remaining nonobstacle free space is considered a legal region to position the mobile robot. Visible tangents are then calculated between all obstacles. A fully connected graph is defined by the obstacle boundaries and all the tangents between them. Determination of a shortest path is accomplished by searching the visibility graph for the lowest cost path between start and goal points. Polygons are commonly used to represent obstacles.

C. CIRCLE WORLD

A simple and effective way to represent obstacles in a configuration space world model is to generate circles that encompass the obstacles. Each obstacle centroid produces the center coordinates of each circle in the circle world model. The initial radius of the circle equals the minimum radius that completely encapsulates the given obstacle. The maximum radius of the moving robot combined with a safety standoff distance is added to each circle radius.

The typical objective of path planning through an obstacle field is to allow robot travel from a known start point to a known goal point. The shortest path is usually desired, and therefore the Euclidean distance traveled is the method used to determine the best route from start to goal. The circle world model and path planning algorithm presented in [Brutzman 92] allows for rapid and efficient determination of shortest-distance paths (Figure 23). The shortest-path planning code in [Brutzman 92] is designed to provide real-time path planning/replanning by autonomous robots. Input to this circle world code is a file that contains a start point, a goal point, a circle centroid

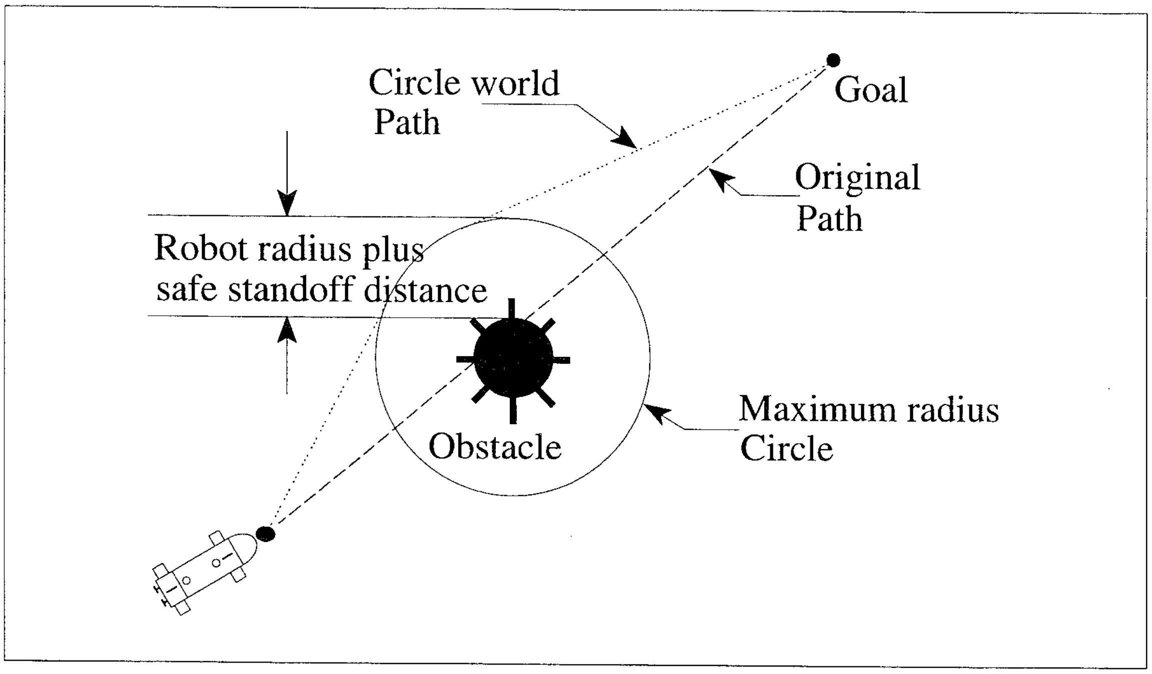


Figure 23. Circle world path created by taking tangents to circles.

location and its radius. The output consists of line segments and arcs that will provide the shortest path around obstacles to the goal. The line segments go from a point to a circle tangent. An arc around the circle to an exit tangent location is included. The exit tangent point of that circle to a tangent on the next circle produces the next segment. This process repeats until the final goal point is reached.

D. SMOOTH MOTION PLANNING

Although the code in [Brutzman 92] provides the shortest path, it does not provide smooth motion planning within this environment. Specifically, the original orientation of the vehicle is not accounted for, nor is the final posture. Technically, the vehicle might stop each time it wants to change direction and reposition itself in the next direction and then move on, but this approach is slow, energy inefficient and unnecessary.

Algorithms used to achieve smooth motion planning are taken from [Kanayama 95]. The objective of applying the smooth motion planning code to the output of the circle world is to provide precise control of Phoenix while allowing rapid travel around obstacles (Figure 24). The hover mode requires Phoenix to maintain position and orientation at a given location. Given the turning radius of a vehicle, smooth motion planning allows the vehicle to go from one point to another along a path that does not require the vehicle to have instantaneous changes in direction. Thus, the vehicle is not required to rotate in place.

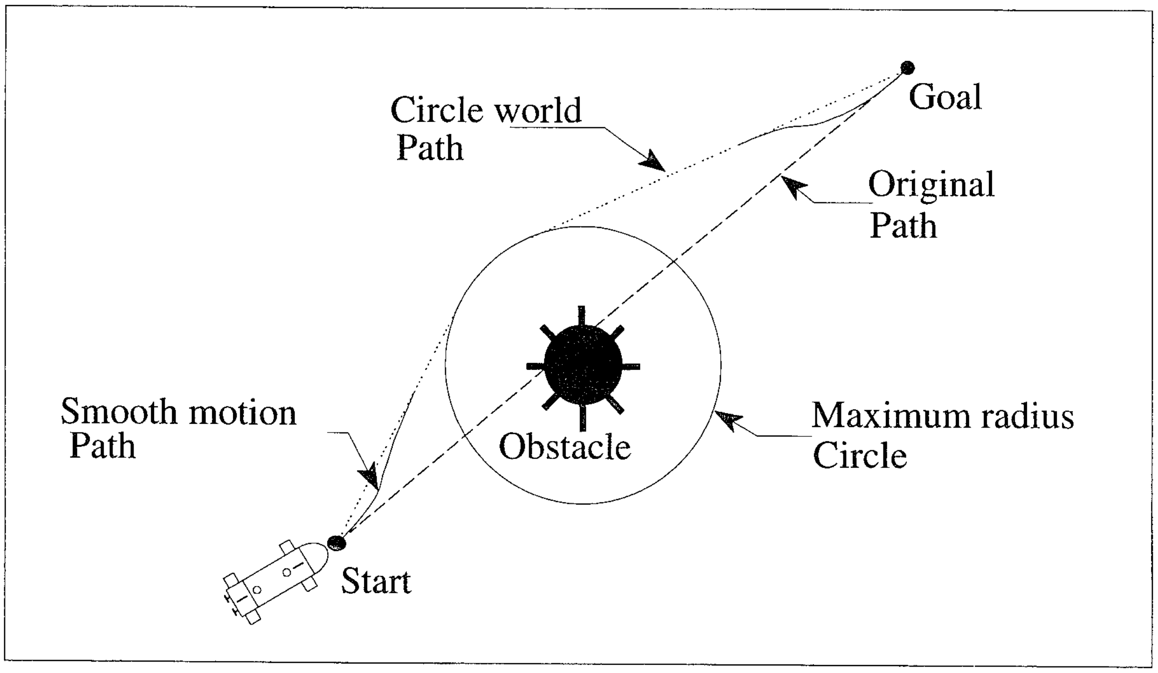


Figure 24. Smooth motion path planning of circle world.

Waypoint control of Phoenix is a method for traversing a path without stopping at individual points on that path. By knowing the turning radius of Phoenix, a smooth course can be planned using waypoint control. Circle world produces two alternating outputs: arcs and segments (Figure 25). Therefore, two methods are necessary to traverse

```
              Circle_World Shortest Path Determination

    Data specifications are according to the AUV Data Dictionary.

     Point    0.00      0.00     0.00      Start

     Point  100.00    100.00      0.00     Goal

     Circle  50.00     55.00      0.00     25.00

     Path                Best path (cost 152.3)

     Segment  0.00      0.00      0.00     67.10      36.77      0.00

     Arc       50.00     55.00     0.00   25.00     313.17   333.80   1 = CCW

     Segment       72.43     43.96     0.00    100.00    100.00      0.00
```

Figure 25. Circle world output file.

these lines. There are two major functions used in the smooth motion planning code to accomplish this requirement. These are described in the following paragraphs.

1. K-spiral

While an AUV is moving forward an instantaneous course change is not possible, particularly given the sluggish response and complex hydrodynamics of AUVs. Circle world provides arcs that are sections of a circle. Circles have a constant rate of change of heading (theta) per distance traveled around its circumference. Circle world provides a tangent to a circle. The tangent is a straight line and therefore the rate of change of theta is a step function where the straight-line segment meets the circular arc. To get Phoenix to smoothly transition from an angular rate of 0 degrees per foot to the rate of the arc it is to travel around, the K-spiral function [Kanayama 95] is used.

The purpose of the K-spiral is to traverse a curve without having instantaneous changes in the rate of change of theta. It provides points around the arc so that the rate of change going onto and off the arc is not instantaneous. The resulting curve closely resembles the original arc except the start and end transitions are smooth and slightly flattened.

2. Line Tracking

Line segments are the other data produced by circle world. A function is necessary to provide a means of smoothly going from a starting posture to a desired directed line. A version of the steering function [Kanayama 95] is used to produce this result. To obtain the solution to this function, several parameters must be known: the curvature of the current path k, the curvature of the desired path k_d, the vehicle's heading θ the desired heading θ_d, and the offtrack distance Δd.

The differences between the actual and desired values determine the error signal dk/ds. Each of these differences, (deltas), is weighted with constants. Creation of the values for these constants is explained extensively in Chapter 5 of [Kanayama 95]. The result is that $a = 3k$, $b = 3k^2$ and $c = k^3$. The constant k is given by $k = 1/\sigma$, where σ has units of length. The larger σ, the smoother the motion. Therefore, σ is known as "smoothness."

The basic steering function is:

$$\frac{dk}{ds} = -(a(k - k_d) + b(\theta - \theta_d) + c\Delta d) \tag{1}$$

To apply this general formula to line tracking,

$$k_d = 0$$

$$\theta_d = \alpha$$

$$\Delta d = -(x - a)\sin\alpha + (y - b)\cos\alpha$$

the curvature is zero, and the orientation is α, if the vehicle is on track with line L. The variable Δd equals the signed distance from the vehicle position $p = (x, y)$ to the directed line L (Figure 26). This signed distance satisfies the condition that Δd is positive if p is on the left side of L, the variable Δd is negative if p is on the right side of L, and delta d equals 0 if p is on L. Thus, the steering function for line tracking becomes:

$$dk/ds = -(ak + b(\theta - \alpha) + c[-(x-a)\sin\alpha + (y - b)\cos\alpha]) \tag{2}$$

Using this steering function, a vehicle will track a line smoothly from a given initial posture.

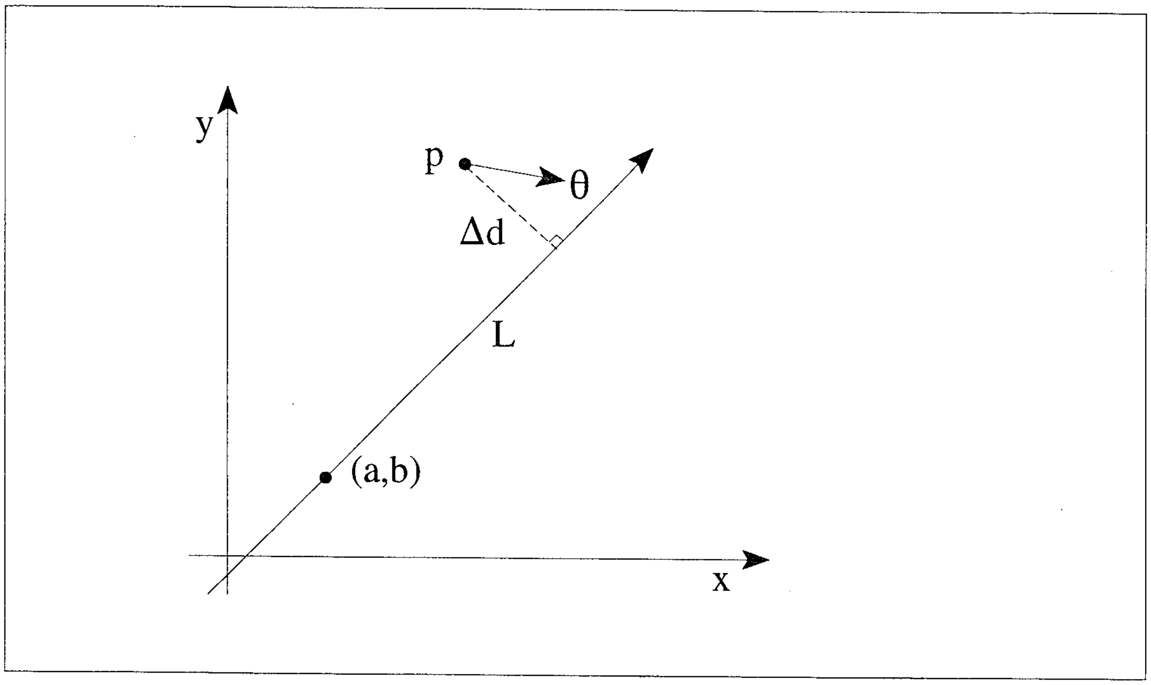


Figure 26. Off-track distance (Δd) and vehicle heading (θ) are used to calculate positional error from line L [Kanayama 95].

3. Assumptions

Several assumptions were made at the outset of the smooth motion portion of this development. First, the circle world results would be 'good-enough' for a submersible vehicle designed for the open ocean. The ability to classify object shapes is still in its early stages and so applying other techniques such as a polygon world is not necessary. The next assumption is that the time to accomplish replanning needs to be minimal. For Phoenix, with its limited processing and slow speeds, this is important and achievable. The next decision was to do line tracking following a K-spiral. Line tracking returns the vehicle to the actual course whereas the K-spiral is an approximate calculation that does not guarantee the final position. If straight compositions from each K-spiral to the start of the next K-spiral were performed, positional errors will accumulate and the difference in final position will be quite noticeable. If Phoenix is traversing a mine field such errors might be fatal. By doing a line track after each K-spiral, it is assured that Phoenix is returned to the desired path.

One of the more difficult parts of the program is deciding the conditions necessary to satisfy the on-line track condition. How much and what type of error is allowable for the program to report it is on course? The decision was made to use the overall error signal produced by (dk/ds) in combination with the off-track error (Δd). These two together provide a good indication that the vehicle is in a stable position. Either error by itself is not sufficient to determine whether or not the vehicle is stable.

4. Program Structure

The basic flow of the tangent-arc smoothing program is to commence a line track from the initial position of the vehicle to the first tangent line segment. Line tracking entails having a smooth path planned from the starting position to somewhere on that line segment. Once the vehicle is considered stable on that line (oriented toward the goal point) the goal point is given as the next point to reach. This goal point will be the intersection point between the line segment and the circle around the first obstacle. When the vehicle reaches this point, it follows a K-spiral path around the arc. Upon completion of the K-spiral, a line track to the next line segment occurs and the cycle repeats.

The only question remaining was how to obtain the final goal point so that the vehicle is oriented in the proper direction upon reaching it. An iterative (hit or miss) approach was originally investigated to find the point at which to leave the last line segment, in order to return to the original goal facing the correct position. However, the concept of 'pretending' that the vehicle was at the goal and to reverse line track to the previous line seemed intriguing. The entire path from the reverse line track was not stored but instead the position that would be used to leave the current line was stored. This was done for several reasons. Reverse tracking (as with any tracking) includes an estimate about where the vehicle may be at a given time. These calculations can not guarantee the exact position when the final line track to the goal occurs, so just the position to commence the straight distance (composition) value was stored. The stored value is used once the on-line motion condition onto the final line has been satisfied.

E. RESULTS

The following is a brief discussion of initial simulation test results. A single small circle that would cause only a small change in the path was first used. Initial results were satisfactory, but errors occurred when larger circles were used. First the K-spiral routine did not work. A path direction clockwise or counterclockwise was needed to traverse the arc in the correct direction. Once this problem was solved, other problems involving the number of iterations of the k-spirals appeared. Not enough points to represent the curve were being used, causing the error to increase. The next bug occurred when the on-line condition was being satisfied even when the vehicle was not on-line. This was due to an incorrect theta calculation. Included in this thesis are gnuplots of two circle cases. The first shows a large circle impeding the vehicle's path. This results in a large distance to reach the initial path line. It then also follows that a large K-spiral was needed to go around the circle. The program successfully tracked around the circle to the goal. This program has been tested in the virtual world and this technique appears to be satisfactory.

F. IMPLEMENTATION ON PHOENIX

The initial application of the replanner on Phoenix amounts to a small variation of the fully functional replanner discussed above. Since there was a tremendous amount of code being generated by several individuals for the current mission, the decision was made early to start with a basic replanning capability (Figure 27). What that meant for the replanner was to insure that Phoenix could avoid obstacles and obtain the final goal point. For the initial implementation, the smooth motion portion was not included.

The key objective in avoiding obstacles was to create a circle large enough around the obstacle so that, if Phoenix was first aimed at the oncoming tangent to the circle and then at the exiting tangent, it would not collide with that object. This resulted in only two additional points in the path (for a single object in the path). Since smooth motion was

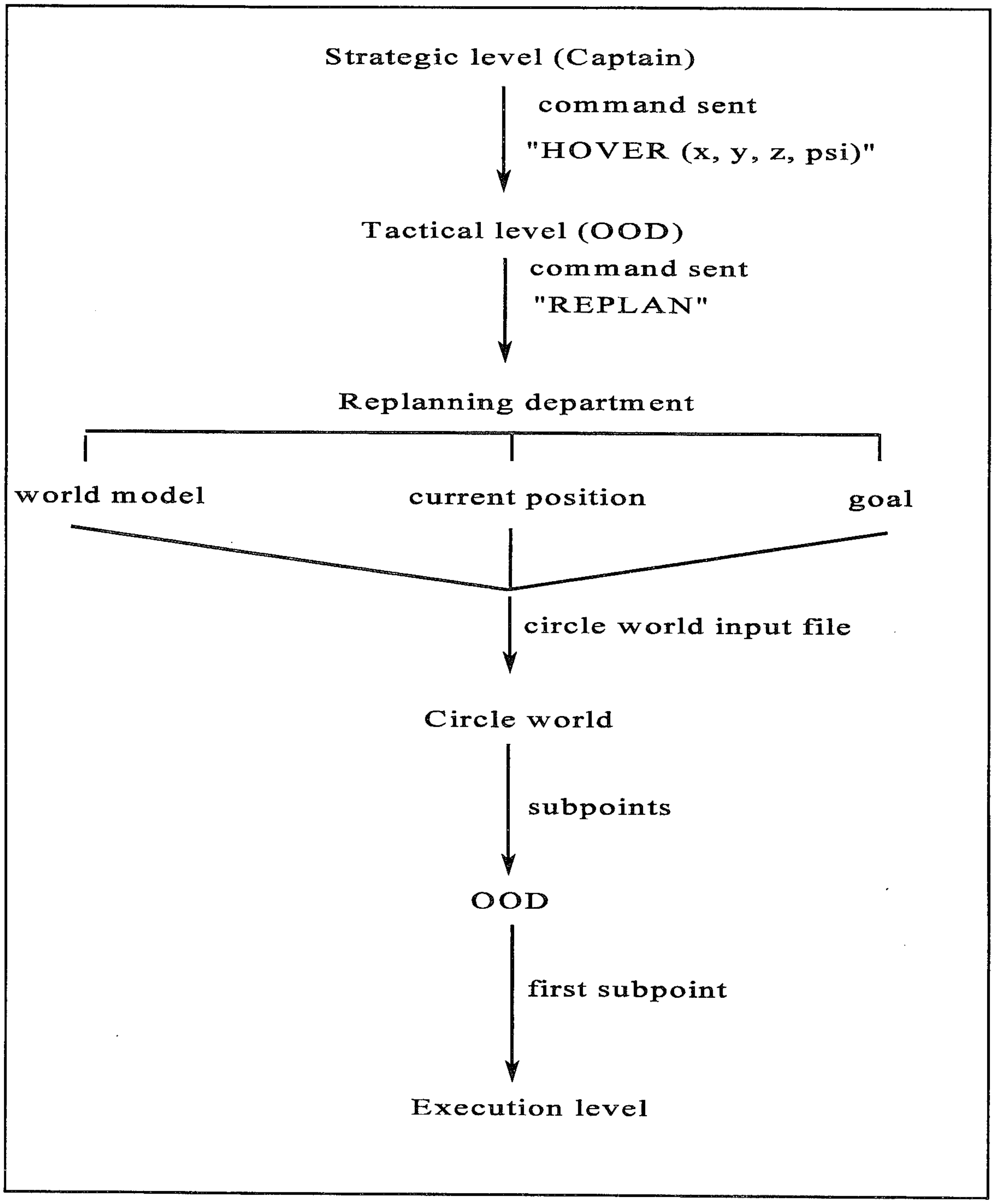


Figure 27. Replanning logic flow path.

not initially included, it was necessary to use hoverpoints for these intermediate points instead of the preferred waypoints. The replanning process can be initiated by several situations, each shown in Figure 27.

When the command to go to a new waypoint or hoverpoint comes down from the Strategic level to the Tactical level OOD, a replan is initiated. The start point is the current ship's position, and the goal is the hoverpoint or waypoint desired. These values are used by the Replanning department along with the most current world model generated by the Sonar department. The world model generated by the Sonar department is a series of circle centroid positions and their respective radii. The Replanning department takes that world model, adds the start and goal points and stores them in a file that is in the format used by the circle world. Circle world is called and passed this file that will then produce a subsequent file containing segments and arcs to traverse the obstacle field. Once the replanning is completed, circle world returns a smoothed shortest path. The Replanning department sends the subwaypoints to the OOD, which in turn are sent to the Execution level to steer the ship.

G. SUMMARY

The underwater environment is a harsh and unforgiving environment. Collisions with obstacles in this environment could lead to the catastrophic loss of the vehicle. Having the ability to path replan is critical for AUV's. Modeling obstacles as circles is an effective means utilized in path replanning given the current state of onboard sensors. Safety ranges are added to the circle that encompasses the objects and producing tangents to the enlarged circles provides a path to circumnavigate these obstacles. Applying smoothing algorithms to the tangents results in a path which can be obtained without slowing the vehicle to make the course changes. The smooth motion planning program allows Phoenix to navigate around obstacles using the waypoint control mode instead of the slower and less-efficient hover mode. A partial implementation of this system has been installed and successfully tested onboard Phoenix.

VI. EXPERT SYSTEM GRAPHICAL USER INTERFACE FOR AUTOMATED MISSION CODE GENERATION

A. INTRODUCTION

The CO of a submarine is a highly knowledgeable, trained and experienced person. It is his responsibility to ensure the mission of the ship gets carried out. He must create a mission plan that is logically sound, feasible and safe before allowing the ship to get underway. It is the CO that makes all the major decisions concerning the operation of the ship and how to safely carry out its mission.

How can all of that knowledge and decision-making capability of the CO be recreated in a program? Will the resultant code be so involved that the creation of a simple mission would require a large amount of time by a computer expert? To have a vehicle that is to be useful in the fleet, it must be easily programmed. Due to the autonomous nature of the vehicle it must contain some type of reasoning capacity. The application of an expert system through a graphical user interface that automatically generates mission code is one solution to the above problems. A system of this type was originally developed by the author and another student as a class project [Davis & Leonhardt 96]. A further elaboration of this system is reported in this chapter.

B. ELIMINATION OF THE EXPERT

An expert system is a computerized system that uses knowledge about some domain to arrive at a solution to a problem from that domain. This solution is essentially the same as that concluded by a person knowledgeable about the domain of the problem when confronted with the same problem [Gonzalez 93]. An expert system is employed for Phoenix in order to eliminate the requirement of having a Prolog coder that is an expert on Phoenix's Strategic level structure. If an individual creates their own Prolog code, there is no automated checking facility to ensure that it is logical, or feasible, or safe. By extensive testing of the expert system, consistent and reliable results are obtained which could never have been achieved using the previous manual method.

C. PROLOG

The top (Strategic) level of the Rational Behavior Model (RBM) defines the overall goals and controls the order of mission execution. The bottom (Execution) level controls motors and actuators. The middle (Tactical) level receives commands from the Strategic level (CO), decides how to accomplish the individual steps of the mission, and then sends commands to the Execution level. Currently, the Strategic level is written in Prolog and interfaces with the Tactical level written in C via foreign function calls (Figure 28). The Prolog language is a partial implementation of predicate logic for computing with added features to increase its practical value as a general purpose programming language. The semantics of Prolog strike a balance between efficient implementation and logical completeness.

```
%Quintus specific to identify the .o file and functions within file.

 foreign_file(tactical, [ood]).

%Calls ood function which is a c function. We send it a string and it
%   returns an integer.

foreign(ood,c,ood(+string,[-integer])).
```

Figure 28. Quintus foreign file call to the *tactical.c* code and ood () function.

Although writing a mission in Prolog is easier than in most other programming languages, it is still a daunting task. Until now it was impossible to define a mission for Phoenix without being thoroughly familiar with how the Strategic level code needed to be structured. The primary goal of this chapter is to describe a system that enables someone with no knowledge of Prolog and minimal knowledge of the AUV control structure to generate sophisticated and reliable mission specifications. The goal is to replace the Prolog/AUV/CO human expert with a simple-to-use computerized code-generating expert system.

D. MISSION PHASES

A mission consists of a series of phases. Each phase has a unique name, the type of phase, follow-on phases and parameters depending on its phase type. There are two disjoint possible follow-on phases: one for a successful mission phase completion, and one for unsuccessful mission phase completion. Currently six phase types are defined: change depth, transit to a point, hover at a point, get a gps fix, conduct a rotating sonar search, and conduct a rotating vehicle sonar search. These phase types were created to accomplish the initial AUV Florida mission. More phases can be installed in future work to provide a basis for programming more complex missions.

Essentially, a mission can be represented as a connected directed graph similar to a decision lattice (Figure 29). Each node represents a phase, and arcs connect a phase

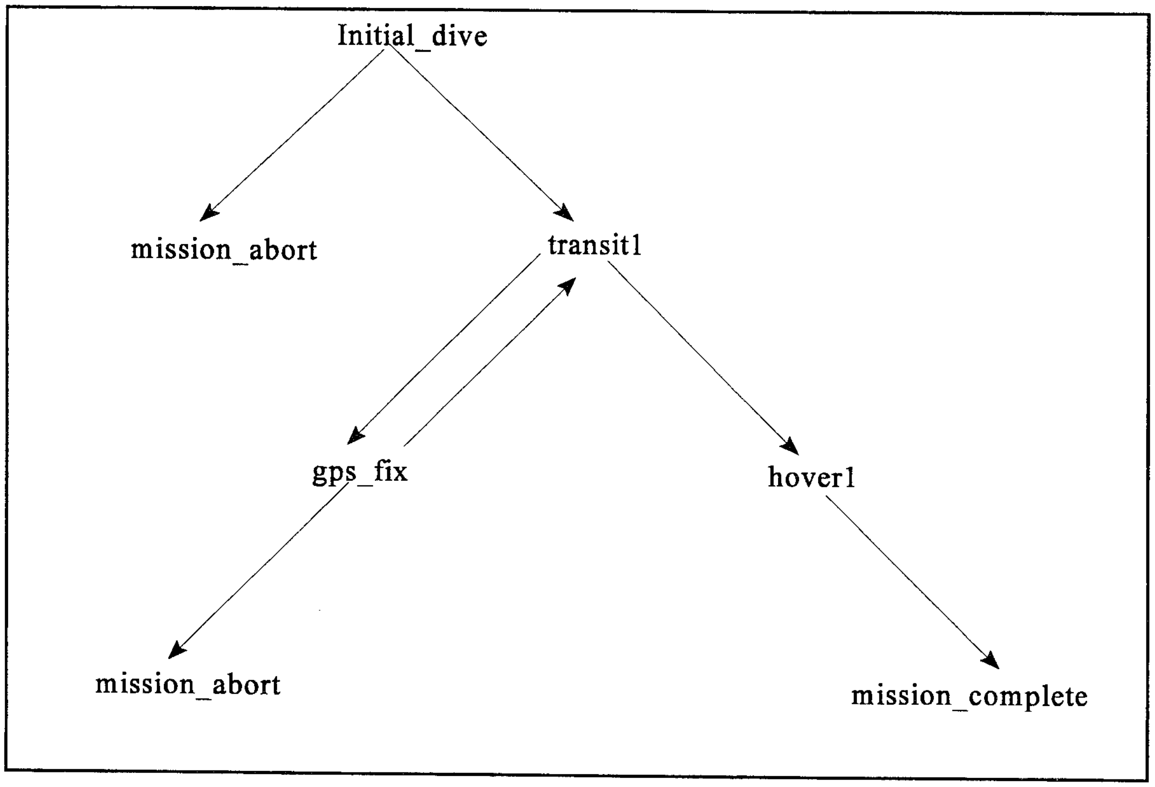


Figure 29. Decision lattice with phase aborts on the left and phase successors on the right.

with its two possible successors. Following any path will eventually lead to Mission Complete or Mission Abort which are flags to the Strategic level that the mission is done.

As previously stated, different types of phases have different types of parameters. For instance a depth-change phase has depth and time-out parameters. Hoverpoint phases have an X position, a Y position, depth, optional heading, and time-out parameters.

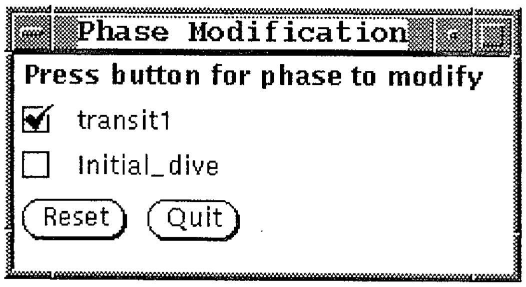
The Strategic level Prolog has two distinct sections: a mission controller (or "doctrine") section and a mission-specific section. Phase execution is controlled by the Prolog mission controller that is the same regardless of the mission (Appendix C). The Prolog code tells the Tactical level what type of phase to execute and what its parameters are. The Tactical level tells the Strategic level when the phase is complete or aborted, at which time the Strategic level decides which phase to execute next (Figure 30).

```
%                Dive to starting depth

execute_phase(1)    :-    nl,printsc('PHASE 1 STARTED.'),nl,
                       ood('depth 2',X),X==1,
                          printsc('DEPTH 2'),nl,
                       ood('start_timer 60',X),X==1,
                          repeat,phase_completed(1).

phase_completed(1) :- ood('ask_depth_reached',X),X==1,
                          printsc('DEPTH REACHED.'),
                          asserta(complete(1)).

next_phase(1)       :-    complete(1),
                          retract(current_phase(1)),
                          asserta(current_phase(2)).

next_phase(1)       :-    abort(1),
                          retract(current_phase(1)),
                          asserta(current_phase(mission_abort)).
```

Figure 30. Depth change phase represents the typical structure of a mission phase.

E. EXPERT SYSTEM SECTIONS

The expert system presented in this chapter can be broken down into three parts: mission planning, mission specification, and mission generation. Mission planning is a help facility that obtains user-specified goals and uses "means-ends" analysis to determine a series of phases that will accomplish the mission [Rowe 88]. Mission specification is the facility that is used to specify the phase graph described above. Phases may also be deleted or modified after they have been specified (Figures 31, 32).



Phase Modification

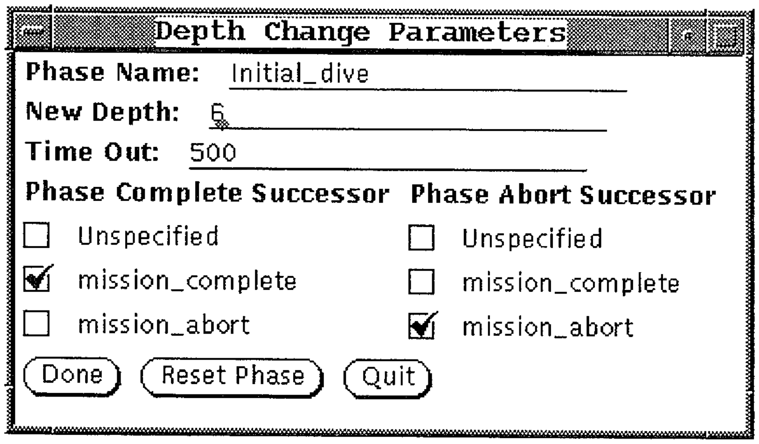
Press button for phase to modify

[✓ transit1]

Initial_dive

Reset Quit

Figure 31. Phase modification window.



Depth Change Parameters

Phase Name: Initial_dive

New Depth: 6

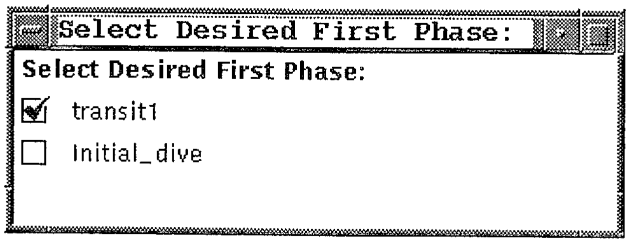
Time Out: 500

Phase Complete Successor	Phase Abort Successor
Unspecified	Unspecified
[✓ mission_complete]	mission_complete
mission_abort	[✓ mission_abort]

Done Reset Phase Quit

Figure 32. Depth change phase window.

The mission generation facility uses the specified graph to generate the Prolog code (including the mission controller) for the specified mission (Figure 33).
All user input is menu based using Prowindows objects, which are usually text items, menus, and buttons. Prowindows enables simple graphics in Prolog [Prowindows 88]. It is not designed for real-time graphics and is therefore quite slow. Clicking on a menu item will cause a fact to be asserted for caching purposes. The values of text items are normally read when their enclosing dialog box is destroyed.


Select Desired First Phase:

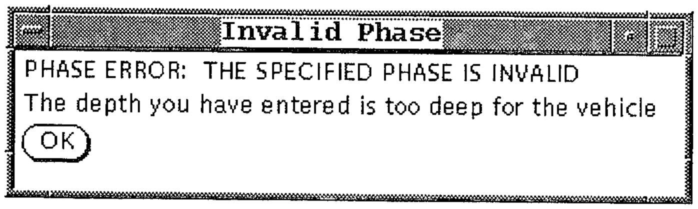
Select Desired First Phase:

✓ transit1

☐ initial_dive

Figure 33. Starting phase selection window.

A rule-based system is then used. Rules are an important knowledge representation paradigm. Rules represent knowledge using the IF-THEN format. The IF portion of a rule is a condition, or premise, which tests the truth value of a set of facts. If these conditions are found true, the THEN portion of the rule is inferred as a new set of facts. Backwards-chaining reasoning is used in Prolog, and involves an examination and application of rules. This type of backward reasoning starts with a desired conclusion and decides if the existing facts support the derivation of a value for this conclusion. As a result, backward reasoning corresponds very closely to depth-first search. There are two forms of rule based error checking within the program. First is the phase checking. If the user enters an invalid phase (depth too deep for the operating area, point outside of operating area, etc.) an error message is generated and the phase is not accepted (Figure 34).


Invalid Phase

PHASE ERROR: THE SPECIFIED PHASE IS INVALID

The depth you have entered is too deep for the vehicle

Figure 34. Phase error window.

The second type of error checking is mission checking. Before code generation the specified phase graph is checked for various possible errors such as loops, unreachable nodes, insufficient time outs, and others. Again, an error window is generated, and no autocode is generated. With either type of error the user can add, modify, or delete phases as necessary to complete the mission specification. Specified phases are stored in memory as facts (Figure 35). These facts are used by the mission error checking routine and by the code generation routine. The code generation routine translates the phase facts into strings and stores them in a file. When all of the phases have been translated, a UNIX shell command is used to invoke a C program which generates and writes Prolog code to the final output file. The code generator is written in C because of the ability to format the output. This makes for a more readable Prolog mission file (Appendix C).

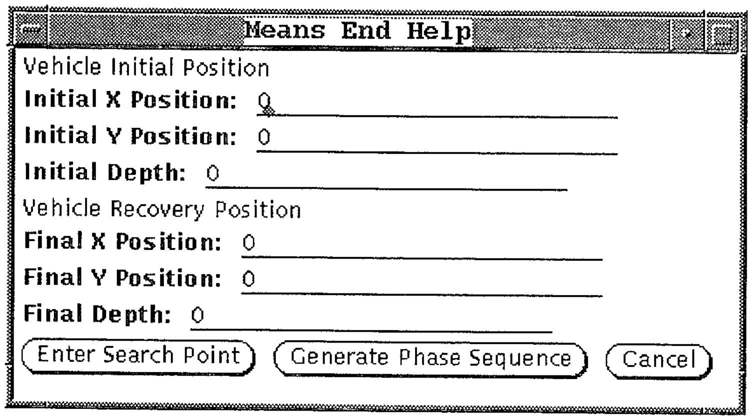
Phase type	Phase name	Complete successor	Abort successor	Time out	Parameters
waypoint	transit1	mission_complete	mission_abort	500	10 10 6
depth_change	Initial_dive	transit1	mission_abort	500	6

Figure 35. Expert system phase facts are stored in a file and then sent to the code generation program for creation of a Prolog mission.

F. USING THE EXPERT SYSTEM

Use of the automated code generator program requires little knowledge of the system and no Prolog experience. The main menu consists of an options menu, a map, a current map menu, a file name text item, and buttons. The map options menu is used to cycle between available operating areas. As different operating areas are specified, the appropriate map and all of the area information is loaded into memory. The options menu is used to add a phase, modify a phase, delete a phase, or obtain means ends help. The first three options pertain to mission specification, while the last accesses the mission planning facility.

Upon clicking the Means-Ends Help menu option, a dialog box is displayed with text items for the AUV starting position and ending position (Figure 36). Both will be filled in based on values from the vehicle information file that loads automatically, but can be changed if needed. Additionally, clicking the Specify Search Point button will display a menu that can be used to display a point that the user wants the AUV to search. The X and Y positions of search points can be specified by clicking on the appropriate


Means End Help

Vehicle Initial Position

Initial X Position: 0

Initial Y Position: 0

Initial Depth: 0

Vehicle Recovery Position

Final X Position: 0

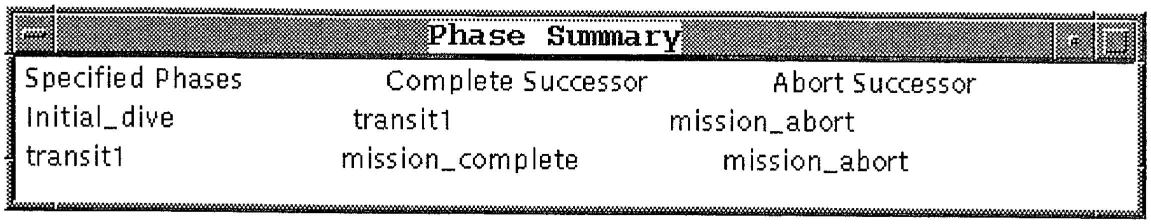
Final Y Position: 0

Final Depth: 0

Figure 36. Means-end planning specification window.

point on the area map in the main dialog box. After specifying the start, recovery, and search points, the system can generate a series of phases that can be specified to achieve the specified goals. Preconditions and postconditions are specified within the program to force certain characteristics. For instance, a GPS fix will be recommended before a search. The mission computed by the means-ends helper is only a recommendation. No code will be generated until the user specifies phases, and there is no requirement to specify the recommended phase sequence.

Mission specification is done by specifying phases one at a time. After clicking the Specify Phase menu item, the user will be asked what type of phase the user wishes to specify. A dialog box with text items and menus corresponding to the required data for the specified phase type will be displayed. The user must specify all of the requested data including the names of the follow-on phases, or an error will be generated. Additionally, the user must provide a name for the phase. The only requirements for the phase name are that there are no spaces allowed in a phase name, and duplicate phase names are not allowed. If the follow-on phases have not yet been specified, the system will prompt the user for the names using the assumption that these phases will be specified later. Once a phase has been specified, it can be modified in the same manner in which it was specified (previously specified data will be shown instead of default data when the data input dialog box is brought up). Phases can also be deleted. Phases can be specified in any order since all of the phases must be specified before code is generated (Figure 37).



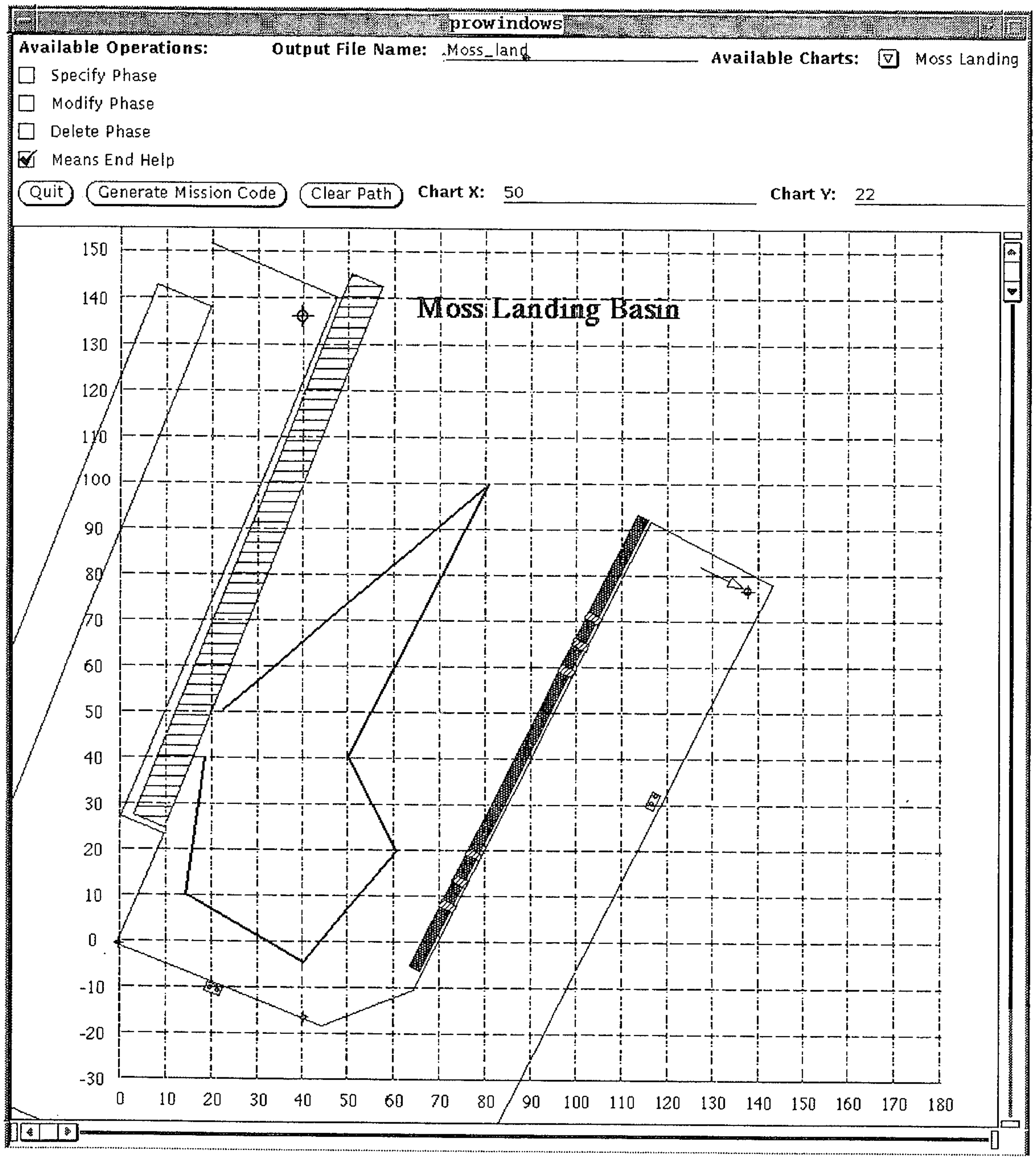
Specified Phases	Complete Successor	Abort Successor
Initial_dive	transit1	mission_abort
transit1	mission_complete	mission_abort

Figure 37. Phase summary window.

Code generation is accomplished after all of the phases have been specified by clicking the Generate Code button (Figure 38). This will cause the graph parsing routine to be invoked and any errors to be reported to the user. If there are no errors or omissions, a UNIX shell command is invoked to call the C program that actually generates the output file containing the Prolog code.

G. SUMMARY

Generation of missions for Phoenix has been done manually by individuals who are experts with the vehicle and the Prolog language. There was no automatic checking of validity of phases or the overall mission. Creation of the expert system graphical user interface for automated code generation has advanced mission planning greatly. No longer is a vehicle expert required nor does the user need to understand Prolog. This system performs numerous checks to ensure phase and mission validity and will automatically produce executable Prolog code which can be used onboard Phoenix.



prowindows

Available Operations: **Output File Name:** Moss_land **Available Charts:** Moss Landing

- [ ] Specify Phase
- [ ] Modify Phase
- [ ] Delete Phase
- [✓] Means End Help

Figure 38. Graphical user interface main menu using Moss Landing chart.

VII. DESIGN AND RESULTS OF EXPERIMENTS

A. INTRODUCTION

Accomplishment of this research was a long process encompassing many months of effort by several individuals. The process started by writing and testing code using the virtual world simulator. Once this phase was completed, testing in the CAUVR test tank occurred, followed by ocean testing at Moss landing Harbor. The last phase was to step back and perform testing in the NPS swimming pool (Figure 39).

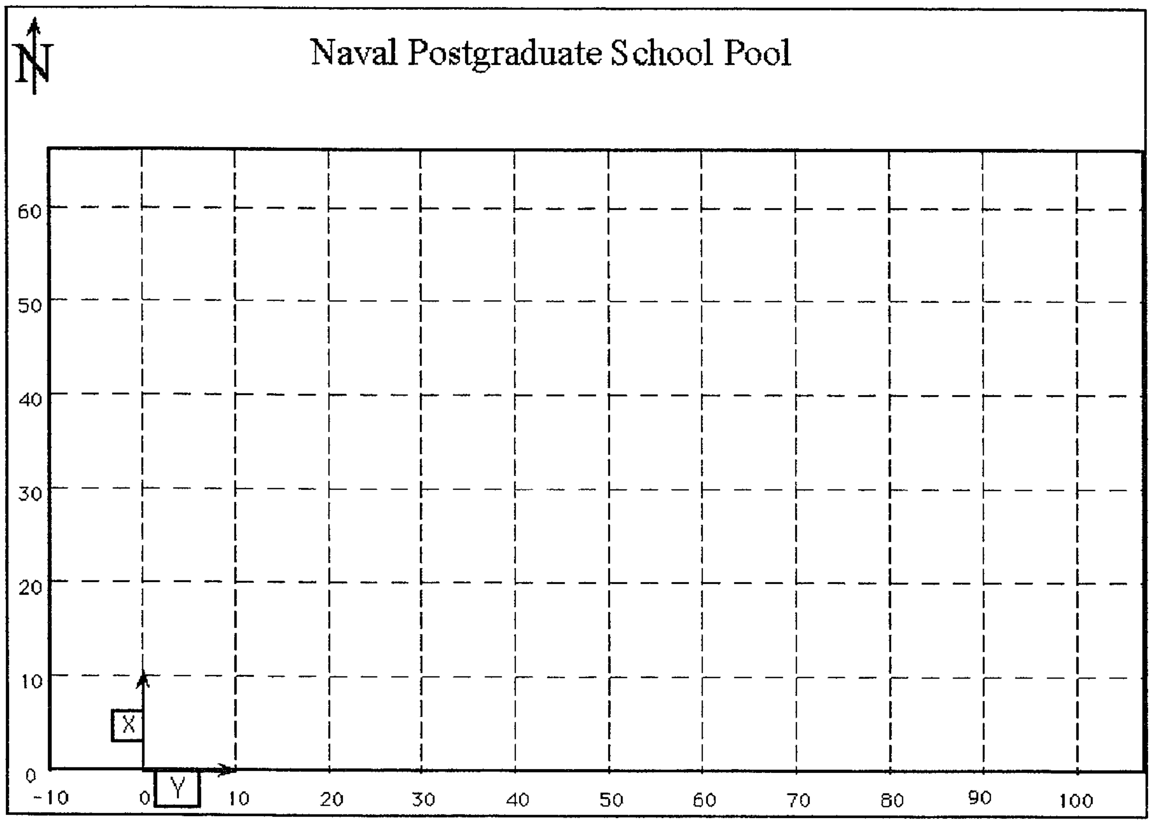


Figure 39. NPS pool chart used for mission planning.

B. SIMULATION TESTING

Writing and testing code on the virtual world simulator was an effective means of generating a lot of new functionality in a relatively short period. An Execution level was already available for use. The Tactical level evolved over several generations, but a sound basic structure was created early so that others might use it to code their department and test while the OOD section was being upgraded. Due to the large amount of parallel coding efforts occurring while the OOD section was being upgraded, it was essential to maintain a valid working version at all times. The most recent correct version of a complete build is maintained on the Internet and can be down loaded at any time (Appendix E).

1. Existing Programs

To plan a structure for the Tactical level, a complete understanding of the capabilities of the existing Execution level code was required. The original Execution level accepted commands and returned a telemetry string containing data from the Execution level. Running the virtual world simulator with the Execution level alone allowed an understanding of its capabilities to be developed.

A good understanding of the Strategic level already existed from class project work [McGhee 95]. The interface between the Strategic and Tactical levels also was clarified by this work. It was understood from the beginning that the Strategic level can only send commands and queries, and the Tactical level can only respond with yes or no to each of the commands or queries.

2. Tactical Level Implementation

The Tactical level was somewhat mysterious at the start of this research since it had never been fully implemented. Writing the LISP simulation provided an excellent basis for its foundation [Leonhardt 95]. The RBM software architecture [Byrnes 93] was the guiding framework for program development. Creation of a basic structure was completed which provided interfaces for the other Tactical level programmers. The basic structure developed was a main program that contained an infinite loop. Before entering the loop, the pipes for communications were created and the departments themselves were forked. Inside each of these departments an infinite loop exists and a means is established for reading and writing to the previously created pipes. Once this structure was in place, each new department had a template for their code.

Communicating with the Execution level became a trivial matter through use of BSD-compliant socket communications. This code was already available and being used by the simulator. This code was quickly incorporated and testing the Tactical level with the Execution level (and therefore virtual world simulator) commenced. A function was created to simulate a mission file. It was indexed and a counter was used to keep track of the next line to read. Once these commands were read, they were sent to the Execution level and to each department. Responses from the Execution level were developed to inform the Tactical level when certain commands were accomplished. An example is the response STABLE HOVER. If the Tactical level sends a HOVER command to the Execution level, it will reply with the above response when the conditions for attaining that point are achieved. Once this response is received, it triggers the Tactical level to read another command from the mission file. With all of this in place, missions could be created to match those written by the Execution level and would appear the same in the simulator.

The transfer of control for mission execution has now been shifted to the Tactical level. Incorporating the departments occurred gradually. When a department was ready for inclusion, the interface was discussed and implemented. Interfacing was relatively easy as each department had a core structure already created with communication channels available. Pertinent functional sections were then transferred to the individual departments code from the main code.

3. Addition of Strategic Level

Adding the Strategic level into the control loop was accomplished in just a few days. Following that implementation, control of the mission now resides where it belongs, at the Strategic level. Initial programming required Tactical level modifications to send responses to queries and commands sent from the Strategic level. The Tactical level version for use with the Strategic level required that main () be removed. For a while, as previously described, two separate versions of the Tactical level were maintained to support operations with and without the Strategic level. A breakthrough occurred when a single version of the Tactical level was created that worked either with or without the Strategic level. All of this code was tested and verified using the virtual world simulator. The project would not have gone very far without this valuable tool. Any project of this magnitude must have a viable simulation model available that runs the actual code that will run onboard the vehicle (Figure 40). The overhead associated with in-water testing is too great and code development would at best be a fraction of what it is to date without use of this simulator.

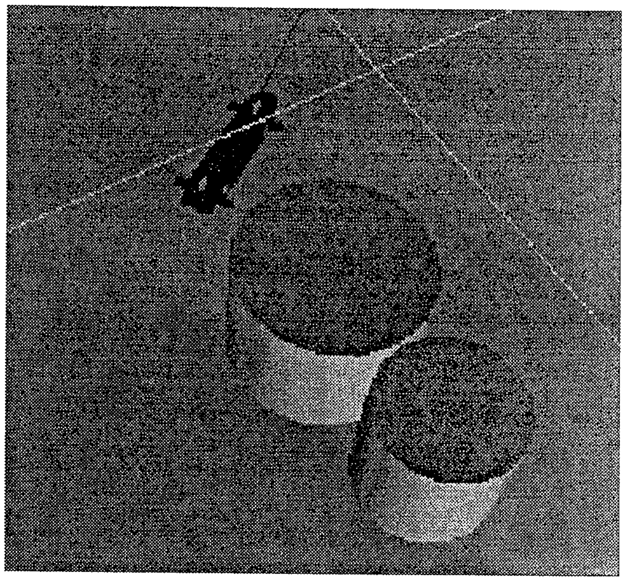


Figure 40. Verification of replanning code using the virtual world.

Simulation is an important step in the development process, but there is no replacement for actual in-water tests. Preliminary testing in the CAUVR test tank was performed and then testing progressed to salt-water testing in Moss Landing. The results of the salt-water testing were discouraging for many reasons.

C. TEST TANK TESTING

Test-tank testing proved that the new software was capable of controlling Phoenix in a stable manner. However, the dead reckoning mathematical models used at the Execution level for determining its position were poor and thus the vehicle was not able to navigate precisely with them. Depth control, lateral control and heading control were exceptions as they functioned with high precision. DiveTracker would not work in the test tank due to the tank's small size and sonar reverberation in the tank. DGPS/GPS signals were also not available in the test tank due to the tank residing in a building. The conclusion obtained from test tank runs was that further testing where DiveTracker could be used and DGPS/GPS would be available, all in a facility that was large enough to obtain water speed calibration was needed. It was decided that calibration of the water speed sensor at Moss landing might help correct the problems encountered in the test tank. However, it was quickly learned that actual in-water time was limited by the Voyager battery life of less than one and a half hours.

D. MOSS LANDING PHASE

Monterey Bay Aquarium Research Institute (MBARI) at Moss Landing California offered the use of their pier facilities until their new ship, the Western Flyer, arrived. This gave two weeks to accomplish salt-water testing. Testing the vehicle at Moss Landing initiated a new era for Phoenix. This represented the first time Phoenix was operated in salt water and also the first time that it would be run autonomously in open water. Testing at Moss Landing proved to be a stressful time due the large number of hardware and weather problems (Figure 41). The majority of the days it rained, and the computer area was wet and cold. Phoenix was not being cooperative and tested the wills of those participating. Several items had to be repaired, and by the time the items were fixed the Voyager battery would die. Zero visibility existed in the water. A physical pointer (a short mast) was affixed to the vehicle so that its location might be known even though the operational depth was only two feet. Early on, DiveTracker was shutting off causing an automatic abort sequence to initiate on Phoenix. Updated DiveTracker software was loaded and the problem was corrected.

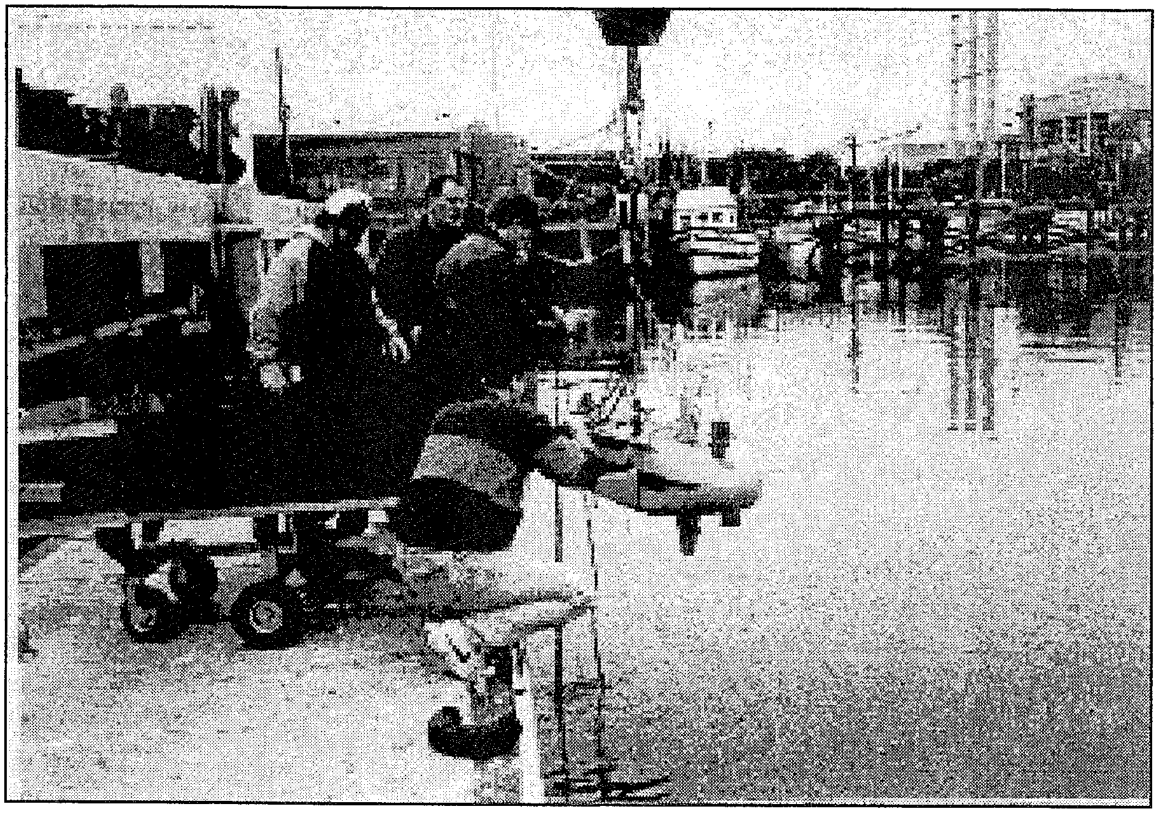


Figure 41. Moss Landing testing. Launching of Phoenix for salt water testing of RBM using only on-board computers and systems.

Another issue that had to be overcome was the use of DGPS/GPS. When the vehicle would initialize at the surface, DiveTracker data would not be available due to the transponder being located on the topside of Phoenix. This meant the only type of fix the Navigation department was obtaining was DGPS/GPS. Although both types of fixes we obtained pierside, once the boat was in the water only the less accurate GPS was available. The range error for GPS is 200 feet and therefore the initial position was not very good. To correct for this once Phoenix initially dove, the vehicle would not move for 30 seconds, allowing DiveTracker to more accurately fix the vehicle's position before attempting to move to the first hoverpoint (Figure 42). The Kalman filter steadied out after this 30 second period and ship's position estimate was fairly accurate [McClarin 96].

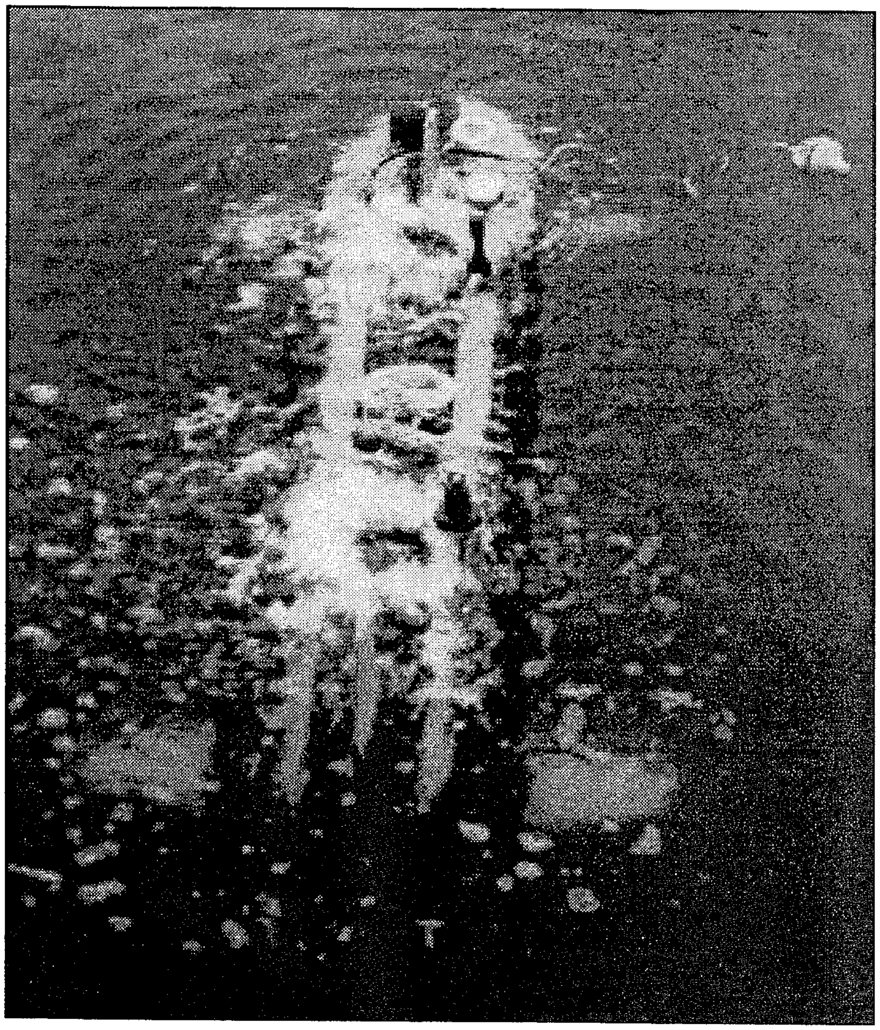


Figure 42. Phoenix submerging at Moss Landing.

What was not achieved during the first week and a half of testing was the ability to obtain a hoverpoint. The reasons for this failure are not absolutely known, but some contributing factors include the water speed sensor not being calibrated, control constants not being adjusted correctly, a strong freshwater/saline layer at 1.5 feet due to recent storms, variances for the navigation filter adjusted too tightly, and most importantly the ability to keep Phoenix running to perform in-water testing. Most of the time at Moss Landing was spent trying to groom these values and fix Phoenix. On the last day of testing the vehicle was capable of getting within 2.5 feet of the hoverpoint. An attempt to run the full Moss Landing mission untethered was attempted. Frustratingly, the vehicle stopped working before obtaining the first hoverpoint and all voltages applied to the motors and controllers remained constant resulting in Phoenix performing a large loop. Unfortunately this loop led it near some concrete pilings and the safety rope attached to Phoenix had to be used to bring it back to the pier.

One final run was attempted. MBARI's new ship was in the pier so the DiveTracker stations and the launch point were moved to behind this ship. Phoenix appeared to start in the correct direction but continued to head out toward open water. After pulling Phoenix back it was determined that Phoenix had fixed its position erroneously from the new DiveTracker locations and thought it was heading in the opposite direction. This error was a valuable result and has been corrected in the current navigation filter [McClarin 96].

E. SUMMARY

Validating code in the virtual world was a long and involved process. Although it took many hours to create a working version, a working version was always obtained. What makes this so important is that the virtual world does not run on batteries and does not require several individuals to run it, so several continuous hours of troubleshooting on the virtual world was possible which would have taken several days with the vehicle. Development of this code without the simulator would not have progressed past the initial stages. The Tactical level OOD code and Replanning code worked extremely well in the virtual world testing.

Testing of the software with Phoenix was a frustrating and a somewhat fruitless experience. The vehicle constantly suffered hardware problems and the lack of adequate testing facilities and support personnel greatly hampered the testing efforts. Nevertheless, all three levels of the RBM were run onboard Phoenix in a salt-water environment, and the viability of both the basic software architecture and its hardware support on-board Phoenix were validated. Lessons learned and corrected problems were the major results of this first open-water test period. More dramatic successes appear to be within grasp.

VIII. RECOMMENDATIONS AND FUTURE WORK

One of the hardest parts of performing work with Phoenix is knowing when to say when. Because this research has advanced Phoenix into the autonomous realm and out of the test-tank environment, much work was done and yet much more work is still needed. The research of this thesis represents just a beginning. Since most of the code was written to accomplish a basic mission, refinement was not necessarily a requirement. Many recommendations for future work are possible.

A. OUTDOOR TEST FACILITY

The current 20ft by 20ft indoor test tank has been outgrown. DiveTracker testing, DGPS/GPS testing, underwater communications testing, sonar classification, etc., all require a larger and/or outdoor facility. MBARI's dock was borrowed for two weeks to do testing before their new ship came in. The trip to MBARI was 30 minutes each way and the schedule was restricted by MBARI's schedule as to when work was permitted. Next, the NPS swimming pool was borrowed to conduct further tests. A deadline as to how long it would be available was also given. A wastewater treatment facility that has been unused for several years is available if the funding to perform the necessary modifications could be obtained. It is located on NPS grounds and provides the size and outdoor requirements to continue the research. Without a facility of this type, further research of AUV's by NPS will be greatly hampered. It is recommended that the Navy fund conversion of the tank to support AUV testing.

B. REPLANNING DEPARTMENT

An obvious example of code needing further refinement is the Replanning department. Trying to accomplish the planned mission took up to the very last week of this thesis research. Due to several other major areas requiring attention, implementation or testing of the smooth motion planning was never achieved onboard Phoenix. The source code has been written to implement this capability, but time prevented its full application. The current program will not function correctly if it is given a path that cannot be achieved smoothly. Specifically, if the vehicle is required to perform a movement within its critical radius, the program is incapable of accomplishing this and goes into an infinite loop. Several simple fixes are available to correct this. First an initial calculation can be performed in the code to determine if the vehicle is inside the critical radius and if so call a separate function that would calculate hoverpoints in place of the waypoints produced by the smooth motion code. A less glamourous solution would check to see if an infinite loop condition exists and return an error message back to the OOD. A further refinement is to include checking for obstacles at waypoints or hoverpoints. Currently, if an obstacle exists at a start of goal point, the program locks up and will cease functioning. A small amount of additional work in this area will reap large dividends.

C. ENGINEERING DEPARTMENT

The Engineering department is a key component of the Tactical level. Currently, casualty procedures at the Execution level are relied on to carry out actions in case of a systems fault. These faults need to be caught early enough so that a mission abort may not be required. Since the Execution level is concerned with real-time, it is up to the Tactical level to perform evaluation and diagnostics to determine if system degradation has occurred. Items to be checked include: battery voltage, fin displacement versus ordered parameters, depth monitoring, thruster voltages versus ship movement, etc. Once the Engineering department is created, a new query is available for the Strategic level. The query "are_critical_systems_ok?" can be added to each phase in the same format as the "ask_time_out" is currently installed. A flag can be set at the Tactical level that can respond to this query based on the output of this department. This can allow AI programs in the Strategic level and in the Engineering department to determine corrective actions [Byrnes 96].

D. MISSION GENERATOR

The expert system for mission generation is currently being upgraded by another student [Davis 96]. Some ideas being considered in this work are code generation directly from means-end analysis, graphical representation of the various means ends solutions, and creating a Strategic level written in C++. A graphical means of viewing how the mission planning process is progressing (tree structure) would greatly enhance the ease of mission generation. Making this system usable from the Internet for mission generation anywhere in the world might be an excellent addition to this system. A web-based home page interface is a worthwhile goal.

E. VXWORKS

The list of what can be done with VxWorks is extensive. The entire vehicle should use VxWorks as its operating system [Wind River Systems 93]. Implementation of the Tactical level under VxWorks is now possible, given the ability to autocode the Strategic level in C++ [Davis 96]. The Execution level would benefit most from the conversion, yet it would require the most work to convert. Writing the device drivers to control the various hardware devices hooked to the Execution level computer would take some time. VxWorks supports Ada, and as a military project that may be of some benefit to pursue. Preliminary work in this direction is reported in [Holden 95]. A further discussion of VxWorks is provided in Appendix D.

F. HARDWARE

Because Phoenix does not have an inexhaustible supply of funding and personnel, changes in Phoenix design are gradual. A high priority improvement should be to replace the GESPAC computer with a more modern and faster system. Such an upgrade should involve substituting (at least) a 68040 processor for the 68030 currently installed. Specifically, the Execution level now has to be slowed to under 7 Hz with the new software due to 68030 processor limitations. It is generally understood that 10 Hz is the minimum desired speed for stable control. Use of a 68040 processor will likely allow this rate to be regained. Software profiling might identify specific bottlenecks.

Experiences at Moss Landing have lead to one major recommendation: reduce the amount of plastic used for connectors. The external Ethernet connector is primarily made out of plastic and is easily sheared off. The run plug which is constantly being put on and removed has plastic threads that eventually break off. The gears used for the fins are mostly plastic and several gears were found to have chipped teeth. It got to the point where piecing together gears was necessary to obtain one good assembly.

A variable ballasting system needs to be developed if this type of vehicle is going to be used in a real world environment. Inside a test tank the ballast of the vehicle does not need to be adjusted much, but in an ocean environment significant changes occur. Changes in ballasting requirements at Moss Landing required 10 lb. adjustments between different runs even at depths less than five feet. Phoenix was not able to go below 2 feet unless it was made negatively buoyant at the surface. Size restrictions prohibit installation of a tank internally to the current vehicle, but installation of external tanks (perhaps on the sides of Phoenix) with a total of approximately 5 gallons capacity would provide sufficient weight to give the necessary ballast for ocean water testing of up to several feet in depth.

Fortunately, the Voyager computer with all its processes runs several times faster than the Execution level running the GESPAC 68030 computer (approximately 6 cycles at Tactical level to 1 cycle at the Execution level). The Voyager has several serial ports available and a software design that will easily accommodate new departments. Several new hardware items can be easily added to the current system. A digital video camera that can take electronic pictures of items in the water would be of great use in documenting objects discovered in the water (i.e., mines). Some kind of mechanical arm for obtaining samples or planting explosives could be added. Environmental survey equipment such as a thermometer, a salinity meter or water sampling mechanism could easily be added to accomplish another of the canonical FAU missions.

G. SUMMARY

Phoenix has entered a new era in its research. It is no longer tied to external computers to perform its missions. Due to the various areas of research this author was involved in with Phoenix, many recommendations have been made. Of paramount importance is the need for a larger outdoor test facility. Further implementation of the Replanning department will enhance vehicle performance. A shell for the Engineering department has been left in the code awaiting its development. Automated mission generation is a major advancement in AUV research. It is currently undergoing further refinement to include the C++ language as an optional replacement for Prolog. VxWorks is the operating system of the future for Phoenix. Initial implementation will be difficult but, once accomplished, Phoenix will be a much more capable platform. Hardware problems with Phoenix, as with any experimental platform, are an inevitable part of research. Several improvements such as elimination of high wear plastic parts, stronger watertight connectors, and a variable ballasting system have been discussed.

Now that Phoenix has obtained an autonomous condition, its capabilities are restricted only by the imagination of scientists and researchers coming up with new roles, missions and solutions. The author hopes that this thesis will help to stimulate interest in other students to continue this work.

APPENDIX A. TACTICAL.C SOURCE CODE

This is main Tactical level code that sets up communications between the Strategic, Tactical and Execution levels and within the processes at the Tactical level. It controls the flow of information and provides an interface for adding additional software modules.

```
/*****************************************************************************/
/*
 Program:         tactical.c  AUV tactical level program

 Authors:         Brad Leonhardt

 Revised:         19 March 96

 System:          SUN Voyager Solaris 2.4 OS; SGI Irix 5.3
 Compiler:        Sun C; IRIX cc

 Compilation:    make [tactical] [strategic] [warnings] [clean]

****************************************************************************
The following section is for running the tactical Prolog code.
****************************************************************************

get all .o files into tactical directory (that is those from execution)

voyager# make
         prolog
         [mission].
         execute_mission.

****************************************************************************

 Execution:
   This is a sample of how to execute the virtual world with the tactical level.
   This assumes you are in Rm 218 Melville on the auvonyx computer.
   There are very few computers that can run the viewer so stick with auvonyx
   for running viewer.  Other code will run on other machines, but you can not
   run all code from the same machine.  Words in [] are optional and [] must be
   removed when using them.  If you want to change your code and remake the
   tactical.c file COPY IT to a different name [tactical.11_15] then make
   changes to code and type > make tactical.

     auvonyx:auv/auv-uvw>          viewer
     auv4d:auv/dynamics>           dynamics [trace]
     auv4d:auv/dynamics>           1
     indigo1:auv/brad/ood/tactical> tactical
     indigo2:auv/execution>        execution virtual auv4d tactical indigo1
                                   [silent] [noemail] [nopause] [trace]
 Description:     tactical level code for Phoenix AUV
 Bugs: Code will hang if Circle Replan occurs ontop of circle
 Active changes:  All of it                                                */

/*****************************************************************************/

#define _BSD_SIGNALS
#include <ctype.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>

#include <time.h>

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

#include "../execution/globals.h"
#include "../execution/defines.h"
#include "../execution/statevector.h"

#if defined(TACTICAL_STANDALONE)
#else

/* #include <quintus/quintus.h> */

#endif

/*****************************************************************************/

/* One stream socket is used with adequate throughput                        */
/*   (although two could work, no performance improvement is expected)       */

/* Be careful that you reserve these port numbers to prevent collisions      */
/*    from other processes requesting ports on your system:                  */

#define DISBRIDGE_TCP_PORT     2056 /* disbridge 1.3 program, server & client */

#define NPSNET_MCAST_PORT      3111 /* Mike Macedonia's multicast DIS 2.0.3   */

                                    /* NPS Autonomous Underwater Vehicle (AUV)*/
                                    /*     Underwater Virtual World (UVW)     */

#define AUVSIM1_TCP_PORT_0     3210 /* os9sender <==> os9server test programs */
#define AUVSIM1_TCP_PORT_1     3211 /* auv execution level <==> virtual world */
#define AUVSIM1_TCP_PORT_2     3212 /* auv execution <==> tactical (telemetry)*/
#define AUVSIM1_TCP_PORT_3     3213 /* auv execution <==> tactical (messages) */
#define AUVSIM1_TCP_PORT_4     3214 /*         port for future use            */
#define AUVSIM1_TCP_PORT_5     3215 /*         port for future use            */
#define AUVSIM1_TCP_PORT_6     3216 /*         port for future use            */
#define AUVSIM1_TCP_PORT_7     3217 /*         port for future use            */
#define AUVSIM1_TCP_PORT_8     3218 /*         port for future use            */
#define AUVSIM1_TCP_PORT_9     3219 /*         port for future use            */
```

```
#define SOCKET_QUEUE_SIZE   5       /* max allowed by TCP/IP                    */

#define MISSION_SCRIPT          "../execution/mission.script"
#define INITIALIZATION_SCRIPT   "initialization.script"
#define SUB_POINTS_FILE         "circle.auv"
#define TELEMETRY_FILE          "tactical_telemetry.output"


/*************************************************************************/
/*            function prototypes                                        */
/*************************************************************************/
/* function prototypes                                                   */
void                set_fl                          ();
/*  uncomment when Engineer Module created
void            Eng_mod                     ();
*/
void            Nav1_mod                    ();
void            Nav3_mod                    ();
void            Sonar_mod                   ();

void            zero_pipe_streams           ();
void            send_telemetry_vector       ();
void            send_command                ();
void            create_pipes_and_forks      ();
int                 open_execution_level_socket     ();
int                 shutdown_socket                 ();
void                kill_pipes                      ();

void                write_to_execution_level_socket ();
int                 read_from_execution_level_socket ();
void                read_from_mission_file          ();
void                read_pipe_streams               ();
void                process_pipe_streams            ();
void            parse_command               ();
void            initialize            ();
void            uppercase                   ();
int                 ood                             ();
void                ood_command_loop                ();
int             command_compare             ();

/* from external_functions.c                                             */

extern void         parse_telemetry_string          ();

/*************************************************************************/
/* dummy function to permit reuse of parse_telemetry_string ()           */

void send_buffer_to_virtual_world_socket ()
{ /* no action */
};

/*************************************************************************/

/* from circtest.c                                                       */
extern void         replanner                       ();
extern void         replanning                      ();
```

```
/* from navigator1.c                                                         */
extern void          navigator1                          ();

/****************************************************************************/

static int                socket_descriptor,
                          socket_accepted,
                          socket_stream;

static int                socket_length = MAXBUFFERSIZE; /*  max packet size  */

static int                bytes_received, bytes_read, bytes_written,
                                          bytes_left, bytes_sent;

static char               command_sent     [MAXBUFFERSIZE],
                          command_received [MAXBUFFERSIZE];

static int                shutdown_signal_received = FALSE;

static char             * ptr_index;

static int                return_state_vector;

static int                socket_already_opened = FALSE;

int TACTICALPARSE = 0;

char Navigator1_string_read[MAXBUFFERSIZE];
char Navigator3_string_read[MAXBUFFERSIZE];
char Sonar_string_read[MAXBUFFERSIZE];
char String_read[MAXBUFFERSIZE];
char String_back[MAXBUFFERSIZE];
char Read_to_clear[MAXBUFFERSIZE];
char command[MAXBUFFERSIZE];
char strategic_command[MAXBUFFERSIZE];
char command_sent2[MAXBUFFERSIZE];
char sonar_file_name[MAXBUFFERSIZE];
char cmd[MAXBUFFERSIZE];
char goal[MAXBUFFERSIZE];

/*  uncomment when Engineer Module created
int Eng_to_OOD_fd[2],     OOD_to_Eng_fd[2],     Eng_telemetry_fd[2],
*/
int Nav1_to_OOD_fd[2],    OOD_to_Nav1_fd[2],    Nav1_telemetry_fd[2],
    Nav3_to_OOD_fd[2],    OOD_to_Nav3_fd[2],    Nav3_telemetry_fd[2],
    Sonar_to_OOD_fd[2],   OOD_to_Sonar_fd[2],   Sonar_telemetry_fd[2];

int time_out = 0;

double tcommand = 0.0;
double time_for_next_command = 0.0;
```

```
int READYFORNEXTCOMMAND    = TRUE;
int WAITINPROGRESS         = FALSE;
int GPSINPROGRESS          = FALSE;
int COLLISION_IMMENENT     = FALSE;
int WAYPOINT_OR_HOVERPOINT= FALSE;
int TACTICALSTANDALONE     = FALSE;
int INITIALIZED            = FALSE;
int INITIALIZING           = FALSE;
int NAV1_INITIALIZED       = FALSE;
int NAV3_INITIALIZED       = FALSE;
int SONAR_INITIALIZED      = FALSE;
int EXECUTION_INITIALIZED = TRUE;
int SUB_POINTS             = FALSE;
int FLIP_FLOP              = FALSE;
int OKTOSENDHOVERPOINT     = FALSE;
int RETURN_VALUE;

FILE * script_file_ptr;
FILE * initialization_file_ptr;
FILE * subpoints_file_ptr;
FILE * telemetry_file_ptr;

double variable1,variable2,variable3,variable4,        /*  parse_command vars */
    variable5;

int variable6,variable7,variable8,variable9;

double ordered_x      = 0;    /* ordered x position                            */
double ordered_y      = 0;    /* ordered y position                            */
double ordered_depth  = 0;    /* ordered depth                                 */
double ordered_course = 0;    /* ordered course                                */


/*************************************************************************/
/* Eng_mod() for future use of engineer diagnostics of hardware          */
/*************************************************************************/
/* uncomment this section when Engineer Module created
void Eng_mod()
{
   TACTICALPARSE = 1;

   for(;;)
    {
         strcpy(String_read,"");
         strcpy(String_back,"");
         if (read(Eng_telemetry_fd[0],String_read,MAXBUFFERSIZE)== -1){}
         else
         {
              parse_telemetry_string(String_read);
         }
```

```
        if (read(OOD_to_Eng_fd[0],String_read,MAXBUFFERSIZE)== -1){}
        else
        {
           if (strcmp (String_read,"QUIT") == 0)
           {
               printf("Terminating Engineer Module \n");
               exit(0);
           }

        }
        read(Eng_to_OOD_fd[0],Read_to_clear,MAXBUFFERSIZE);
        write(Eng_to_OOD_fd[1],String_back,MAXBUFFERSIZE);

      }
}
*/

/*****************************************************************************/
/* navigator1 () implements gps, dgps, and divetracker into a kalman filter   */
/* the outputs are periodic FIXes and possible loss of track info             */
/*****************************************************************************/

void Nav1_mod()
{
   TACTICALPARSE = 1;

   if (TRUE)  /*  change to FALSE if navigator1 () not working or not used    */
   {
      navigator1();
   }
   else
   {
    for(;;)
    {
         strcpy(String_read,"");
        strcpy(String_back,"");
        if (read(Nav1_telemetry_fd[0],String_read,MAXBUFFERSIZE)== -1){}
        else
        {
             parse_telemetry_string(String_read);
        }


       if (read(OOD_to_Nav1_fd[0],String_read,MAXBUFFERSIZE)== -1){}
         else
        {

           parse_command(String_read);
           if (strcmp (command,"QUIT") == 0)
           {
               printf("Terminating Navigator1 Module \n");
               exit(0);
           }
        }
        read(Nav1_to_OOD_fd[0],Read_to_clear,MAXBUFFERSIZE);
        write(Nav1_to_OOD_fd[1],String_back,MAXBUFFERSIZE);

      }
   } /* child process infinite loop, do not return */

}

/*****************************************************************************/
/* navigator3 () implements mission replanning for obstacle avoidance        */
/*****************************************************************************/

void Nav3_mod()
{
    replanning();
}


/*****************************************************************************/
/* sonar () controls st725 and creates world model outputs for use by        */
/* navigator3 (uses line fitting to create world model and outputs circles)  */
/*****************************************************************************/


void Sonar_mod()
{
   TACTICALPARSE = 1;

   if (TRUE)  /*     change to FALSE if sonar () not working or not used      */
   {
      sonar();
   }
   else
   {
     for(;;)
     {
      strcpy(String_read,"");
      strcpy(String_back,"");
      if (read(Sonar_telemetry_fd[0],String_read,MAXBUFFERSIZE)== -1){}
      else
      {
           parse_telemetry_string(String_read);
      }

      if (read(OOD_to_Sonar_fd[0],String_read,MAXBUFFERSIZE)== -1){}
      else
      {
          parse_command(String_read);
          if (strcmp (String_read,"QUIT") == 0)
          {
              printf("Terminating Sonar Module \n");
              exit(0);
          }

      }
      read(Sonar_to_OOD_fd[0],Read_to_clear,MAXBUFFERSIZE);
      write(Sonar_to_OOD_fd[1],String_back,MAXBUFFERSIZE);
      if(strcmp (String_back,"") != 0)
      if(TRACE) printf("INSIDE SONAR MESSAGE :%s\n",String_back);
      strcpy(String_back,"");
    }
   } /*            child process infinite loop, do not return            */
}

/*******************************************************************************/

int open_execution_level_socket ()       /* Initialize communications blocks  */
{
int    TRACE = TRUE;
      if (socket_already_opened == TRUE)
      {
           if (TRACE)
           {
             printf ("open_execution_level_socket (): ");
             printf ("socket_already_opened, returning\n");
           }
           return (-1);
      }
      else if (TRACE) printf ("open_execution_level_socket () starting\n");

/*******************************************************************************/
/* Initialize both client & server *******************************************/

   /* Signal handlers for termination to override net_open () and net_close ()*/
   /*    signal handlers.  Otherwise you are unable to ^C kill this program.  */

#if defined(sun)
signal (SIGHUP,  shutdown_socket);          /*    hangup                      */
signal (SIGINT,  shutdown_socket);          /*    interrupt character         */
signal (SIGKILL, shutdown_socket);          /*    kill signal from Unix       */
signal (SIGPIPE, shutdown_socket);          /*    broken pipe from other host */
signal (SIGTERM, shutdown_socket);          /*    software termination        */
#endif

/*******************************************************************************/
/* Initialize server ***********************************************************/

   /* setup to listen for client to attempt connection                        */
   {
      /* Server opens server port *****************************************/

      /* Open TCP (Internet stream) in socket                                 */
      if ( (socket_descriptor = socket (AF_INET, SOCK_STREAM, 0)) < 0 )
      {
          printf ("open_execution_level_socket() can't 'open' stream socket\n");
          return (-1);
      }
      else if (TRACE)
      {
          printf("open_execution_level_socket () socket 'open' successful\n");
      }

      /* Bind local address so client can talk to server                      */
    #if defined(sgi)
    bzero ((void *) &server_address, sizeof (server_address));
    #endif
    server_address.sin_family       = AF_INET;    /* Internet protocol family */

    /* make sure port is in network byte order                               */
    server_address.sin_addr.s_addr = htonl (INADDR_ANY);
    server_address.sin_port        = htons (AUVSIM1_TCP_PORT_2);

    if (bind (                    socket_descriptor,
              (struct sockaddr *) &server_address,
                          sizeof (server_address)) < 0)
    {
        printf("open_execution_level_socket () socket 'bind' unsuccessful\n");
        exit(0);
        return (-1);
    }
    else if (TRACE)
    {
        printf("open_execution_level_socket () socket 'bind' successful\n");
    }

    /* prepare socket queue for connection requests using listen             */

    listen (socket_descriptor, SOCKET_QUEUE_SIZE);
    if (TRACE)
    {
      printf("open_execution_level_socket () socket 'listen' complete ...\n");
    }

    /* Server 'accept' waits for client connections ************************/

    if (TRACE)
    {
    printf("open_execution_level_socket () socket waiting to 'accept' ...\n");
    }
    bytes_received = sizeof (socket_descriptor);
    while ((socket_accepted  = accept ( socket_descriptor,
                                        &server_address,
                                        &bytes_received)) < 1)
       if (TRUE)  /* blocking code */
       {
         printf ("open_execution_level_socket () socket ");
         printf ("'accept' unsuccessful, sleeping 1 second ...\n");
         sleep (1);
       }
    printf ("open_execution_level_socket () ");
    printf ("connection is open between networks.\n");

} /* end initialization                                                      */

socket_stream = socket_accepted;   /* server */

socket_already_opened = TRUE;
```

```
   if (TRACE)
   {
      printf ("AUVsocket SERVER: socket_descriptor = %d\n", socket_descriptor);
      printf ("                  socket_accepted   = %d\n", socket_accepted);
      printf ("                  socket_stream     = %d\n", socket_stream);
   }
   return (TRUE);
}

/*************************************************************************/
int read_from_execution_level_socket ()  /*using global variables state vector*/
{
   int       start_index, offset, index;

   static int no_bytes_read_count = 0;

   /*********************************************************************/
   /* Receiver block                                                    */
   /*********************************************************************/

   for (index = 0; index < MAXBUFFERSIZE; index++)     /*    uppercase        */
         command_received [index] = (char) 0;

   /* listen to remote host, relay to local network/program                   */

   bytes_left         = socket_length;
   bytes_received     = 0;
   ptr_index          = command_received;

   while ((bytes_left > 0) && (bytes_received >= 0))  /* read loop ************/
   {
      bytes_read = read (socket_stream, ptr_index, bytes_left);

      if      (bytes_read <  0) bytes_received = bytes_read;
      else if (bytes_read >  0)
      {
           bytes_left     -= bytes_read;
           bytes_received += bytes_read;
           ptr_index      += bytes_read;
      }
      if (TRACE)
      {
          printf ("read_from_execution_level_socket () ");
          printf ("receiver block loop bytes_read = %d\n", bytes_read);
      }

      /* if nothing is waiting to be read, break out of read loop              */
      if ((bytes_read == 0) && (bytes_received == 0)) break;
   }

   if  (bytes_received < 0)        /* failure */
   {
        if (TRUE)
        {
            printf ("read_from_execution_level_socket () ");
            printf ("receiver block read failed, bytes_received = %d\n",
                     bytes_received);
        }
        if (no_bytes_read_count >= 5)
        {
            printf ("read_from_execution_level_socket () ");
            printf ("failed, exit.\n");
            exit (0);
        }

        no_bytes_read_count++;
        printf ("no_bytes_read_count == %d\n", no_bytes_read_count);
        sleep (1);
        return_state_vector = FALSE;
        return (return_state_vector);
    }
    else if (bytes_received == 0)  /* no transfer */
    {
        if (TRUE)
        {
            printf ("read_from_execution_level_socket () ");
            printf ("receiver block bytes_received =  0 bytes\n");
        }

        if (no_bytes_read_count >= 5)
        {
            printf ("read_from_execution_level_socket () ");
            printf ("failed, exit.\n");
            exit (0);
        }

        no_bytes_read_count++;
        printf ("no_bytes_read_count == %d\n", no_bytes_read_count);
        sleep (1);

        return_state_vector = FALSE;
        return (return_state_vector);
    }
    parse_command(command_received);

    if (strcmp(command,"AUV_STATE") == 0)
    {
       parse_telemetry_string(command_received); /* update state vector     :) */
       send_telemetry_vector(command_received);
       strcpy(command,"");
       strcpy(command_received,"");
    }
   /* else send_command(command_received);  type mismatch !!<<<<<<<<*/

} /* end read_from_execution_level_socket ();                              */

/***********************************************************************/
/***********************************************************************/
void write_to_execution_level_socket ()  /* using global variables state vector
*/
{
     /* send string:  command_sent */

    if (TRACE)
         printf ("[write_to_execution_level_socket (): command_sent=%s]\n",
                           command_sent);
```

```
/***************************************************************************/
/* Sender block                                                            */
/***************************************************************************/

bytes_received = strlen (command_sent);

if (bytes_received < 0)  /* copy failure                                 */
{
    printf ("write_to_execution_level_socket () ");
    printf ("copy from command_received unsuccessful\n");
    shutdown_socket (0);
    return;
}

else if (bytes_received > socket_length)
     {
        printf ("write_to_execution_level_socket () ");
        printf ("send_telemetry_to_server error: \n");
        printf ("bytes_received too big for packet socket_length");
        printf ("          [bytes_received=%d", bytes_received);
        printf ("] > [socket_length=%d]; ", socket_length);
        printf ("string truncated\n");
     }

bytes_left         = socket_length;
bytes_written      = 0;
ptr_index          = command_sent;

while ((bytes_left > 0) && (bytes_written >= 0))  /*  write loop ***********/
{
    bytes_sent = write (socket_stream, ptr_index, bytes_left);

    if      (bytes_sent <  0) bytes_written = bytes_sent;
    else if (bytes_sent >  0)
    {
        bytes_left     -= bytes_sent;
        bytes_written  += bytes_sent;
        ptr_index      += bytes_sent;
    }



    if (TRACE && (bytes_written != socket_length))
    {
       printf ("write_to_execution_level_socket () loop  bytes sent = %d\n",
                 bytes_sent);
    }
}
if   (bytes_written < 0)
{
      printf ("write_to_execution_level_socket () send failed, %d",
                bytes_written);
      printf (" bytes_written\n");
}
```

```
   else if (TRACE)
   {
      printf ("write_to_execution_level_socket () total bytes sent = %d\n",
               bytes_written);
   }

   return;

}  /* end write_to_execution_level_socket ();                                  */

/*****************************************************************************/
/*****************************************************************************/

int  shutdown_socket (int signal_thrown)                     /* Shutdown block */

{
   int close_result;

   if (TRACE)
       printf ("shutdown_socket (%d) shutdown in progress ...\n",
                signal_thrown);

   shutdown_signal_received = TRUE;

   /* No need to send a message to other side that bridge is going down,      */
   /*    since SIGPIPE signal trigger may cause shutdown on other side        */

   close_result = close (socket_stream);

   if (close_result == -1)
   {
       printf ("shutdown_socket () close (socket_stream) failed,");
       printf ("possibly due to being already closed by other side.\n");
   }

   if      (signal_thrown == SIGHUP)
   {
       printf ("shutdown_socket (%d) signal_thrown == ", signal_thrown);
       printf ("SIGHUP (signal hangup)\n");
       kill_pipes();
       exit (0);
   }


   else if (signal_thrown == SIGINT)
   {
       printf ("shutdown_socket (%d) signal_thrown == ", signal_thrown);
       printf ("SIGINT (interrupt character)\n");
       kill_pipes();
       exit (0);
   }
   else if (signal_thrown == SIGKILL)
   {
       printf ("shutdown_socket (%d) signal_thrown == ", signal_thrown);
       printf ("SIGKILL (kill signal from Unix)\n");
       kill_pipes();
       exit (0);
   }
   else if (signal_thrown == SIGPIPE)
   {
       printf ("shutdown_socket (%d) signal_thrown == ", signal_thrown);
       printf ("SIGPIPE (broken pipe from other host)\n");
       kill_pipes();
       exit (0);
   }
   else if (signal_thrown == SIGTERM)
   {
       printf ("shutdown_socket (%d) signal_thrown == ", signal_thrown);
       printf ("SIGTERM (software termination)\n");
       kill_pipes();
       exit (0);
   }
   if (TRUE) printf ("shutdown_socket () complete\n");
   return (close_result);
} /* end shutdown_socket () */

/******************************************************************************/
/*  kill_pipes()  Graceful means to terminate child processes                 */
/******************************************************************************/

void kill_pipes()
{
    fclose(telemetry_file_ptr);

/*  uncomment when Engineer Module created
    read(OOD_to_Eng_fd[0],Engineer_string_read,MAXBUFFERSIZE);
    write(OOD_to_Eng_fd[1],"QUIT",MAXBUFFERSIZE);
*/
    read(OOD_to_Sonar_fd[0],Sonar_string_read,MAXBUFFERSIZE);
    write(OOD_to_Sonar_fd[1],"QUIT",MAXBUFFERSIZE);
    read(OOD_to_Nav1_fd[0],Navigator1_string_read,MAXBUFFERSIZE);
    write(OOD_to_Nav1_fd[1],"QUIT",MAXBUFFERSIZE);
    read(OOD_to_Nav3_fd[0],Navigator3_string_read,MAXBUFFERSIZE);
    write(OOD_to_Nav3_fd[1],"QUIT",MAXBUFFERSIZE);
    sleep(10);
/*    kill (Navigator1, SIGKILL);
    kill (Navigator3, SIGKILL);
    kill (Sonar, SIGKILL);
*/
}

/******************************************************************************/
/* set_fl(int fd, int flags) Function for setting pipes as nonblocking        */
/******************************************************************************/
void set_fl(int fd, int flags)
{
   int val;

   if ((val = fcntl(fd, F_GETFL, 0)) < 0)
      if (TRACE) printf("fcntl Error\n");

   val |= flags;   /* turn on flags */

   if (fcntl(fd, F_SETFL, val) < 0)
      if (TRACE) printf("fcntl F_SETFL error\n");
}
```

```
/****************************************************************************/
/* initialize() Function for starting up auv (coms, child processes)        */
/****************************************************************************/
void initialize()
{
    int  parameters_read;
    int  read_another_line = TRUE;
    TACTICALPARSE = 1;
    TRACE = FALSE;
  if (TRACE) printf("Inside Initialize\n");
    open_execution_level_socket ();
  if (TRUE) printf("Inside Initialize socket open\n");
    create_pipes_and_forks ();
  if (TRACE) printf("Inside Initialize pipes and forks created\n");

    if (TACTICALSTANDALONE == TRUE)
    {
       if ((script_file_ptr = fopen (MISSION_SCRIPT, "r")) == ((FILE *) 0))
       {
           printf("INVALID INPUT FILE NAME  %s\n",MISSION_SCRIPT);
           fclose  (script_file_ptr);
           exit (0);
       }
    }

    system("cp tactical_telemetry.output tactical_telemetry.output.bak");
    system ("rm tactical_telemetry.output");

    if ((telemetry_file_ptr = fopen (TELEMETRY_FILE, "a")) == ((FILE *) 0))
    {
        printf("INVALID TELEMETRY_FILE NAME  %s\n",TELEMETRY_FILE);
        fclose (telemetry_file_ptr);
    }

    strcpy (command, "INITIALIZE");
    tcommand = t;
    if (TRACE) printf("tcommand : %4.1lf\n\n",tcommand);


    if(TACTICALSTANDALONE == TRUE)
        send_command(command);
    else
    {
        if (TRACE)  printf("send_command \n");
        write(OOD_to_Nav1_fd[1],command,MAXBUFFERSIZE);
        write(OOD_to_Nav3_fd[1],command,MAXBUFFERSIZE);
        write(OOD_to_Sonar_fd[1],command,MAXBUFFERSIZE);
    }

    if (TRACE)  printf("send_command complete.\n");
    read_pipe_streams();
    if (TRACE)  printf("read_pipe_streams() complete.\n");

    process_pipe_streams();
    if (TRACE)  printf("process_pipe_streams() complete.\n");

    strcpy(command_sent, "");
```

```
/* Read initialization commands from initialization.script for the prolog code*/

    if( ! TACTICALSTANDALONE)
    {
       if ((initialization_file_ptr = fopen (INITIALIZATION_SCRIPT, "r"))
              == ((FILE *) 0))
       {
           printf("INVALID INPUT FILE NAME  %s\n",INITIALIZATION_SCRIPT);
           fclose  (initialization_file_ptr);
       }

       while (read_another_line == TRUE)
       {
           if (fgets (command_sent, 120, initialization_file_ptr) != NULL)
           {
              parameters_read = sscanf (command_sent, "%s",command);
              bytes_received = strlen (command_sent);

              if    ((parameters_read != 1)           ||
                      (strlen (command)         == 0) ||
                      (strcmp (command,"#")     == 0) ||
                      (strlen (command_sent)    == 0) ||
                      (command_sent [0]         == '\n'))
                           read_another_line = TRUE;

              else
              {
                  write_to_execution_level_socket  ();
                  write(OOD_to_Nav1_fd[1],command_sent,MAXBUFFERSIZE);
                  write(OOD_to_Nav3_fd[1],command_sent,MAXBUFFERSIZE);
                  write(OOD_to_Sonar_fd[1],command_sent,MAXBUFFERSIZE);
                  strcpy (command_sent, "");
                  read_from_execution_level_socket ();
                  read_pipe_streams();
                  process_pipe_streams();
              }
          }

          else
          {
              fclose  (initialization_file_ptr);
              read_another_line = FALSE;
          }
       }
    }

    if (TRACE)  printf("! TACTICALSTANDALONE loop complete.\n");
    while ((TACTICALSTANDALONE == TRUE) &&
           (INITIALIZED == FALSE)       && ((t - tcommand) < 105.0))
    {
        if (TRACE)  printf("TACTICALSTANDALONE = TRUE\n");
            ood_command_loop();

        if (NAV1_INITIALIZED       == TRUE &&
            NAV3_INITIALIZED       == TRUE &&
            SONAR_INITIALIZED      == TRUE &&
            EXECUTION_INITIALIZED == TRUE)
            {
                INITIALIZING = FALSE;
                INITIALIZED = TRUE;
            }
    }


    if (TACTICALSTANDALONE && INITIALIZED == FALSE)
    {
        kill_pipes();
        sleep(5);
        strcpy (command_sent, "QUIT");
        write_to_execution_level_socket  ();
        strcpy (command_sent, "");
        read_from_execution_level_socket ();
        exit (0);
    }

    TRACE = FALSE;
}

/*******************************************************************************/
/* create_pipes_and_forks()    Establishes pipes and forks processes          */
/*******************************************************************************/

void create_pipes_and_forks()
{
     int Navigator1,Navigator3,Sonar;

/*   uncomment when Engineer Module created
   pipe(OOD_to_Eng_fd);
   pipe(Eng_to_OOD_fd);
   pipe(Eng_telemetry_fd);
*/
   pipe(OOD_to_Nav1_fd);
   pipe(Nav1_to_OOD_fd);
   pipe(Nav1_telemetry_fd);

   pipe(OOD_to_Nav3_fd);
   pipe(Nav3_to_OOD_fd);
   pipe(Nav3_telemetry_fd);

   pipe(OOD_to_Sonar_fd);
   pipe(Sonar_to_OOD_fd);
   pipe(Sonar_telemetry_fd);
```

```
/*  uncomment when Engineer Module created
   set_fl(OOD_to_Eng_fd[0], O_NONBLOCK);
   set_fl(OOD_to_Eng_fd[1], O_NONBLOCK);
   set_fl(Eng_to_OOD_fd[0], O_NONBLOCK);
   set_fl(Eng_to_OOD_fd[1], O_NONBLOCK);
   set_fl(Eng_telemetry_fd[0], O_NONBLOCK);
   set_fl(Eng_telemetry_fd[1], O_NONBLOCK); */

   set_fl(OOD_to_Nav1_fd[0], O_NONBLOCK);
/*   set_fl(OOD_to_Nav1_fd[1], O_NONBLOCK); */
   set_fl(Nav1_to_OOD_fd[0], O_NONBLOCK);
/*   set_fl(Nav1_to_OOD_fd[1], O_NONBLOCK); */
   set_fl(Nav1_telemetry_fd[0], O_NONBLOCK);
/*   set_fl(Nav1_telemetry_fd[1], O_NONBLOCK); */

   set_fl(OOD_to_Nav3_fd[0], O_NONBLOCK);
/*   set_fl(OOD_to_Nav3_fd[1], O_NONBLOCK); */
   set_fl(Nav3_to_OOD_fd[0], O_NONBLOCK);
/*   set_fl(Nav3_to_OOD_fd[1], O_NONBLOCK); */
   set_fl(Nav3_telemetry_fd[0], O_NONBLOCK);
/*   set_fl(Nav3_telemetry_fd[1], O_NONBLOCK); */

   set_fl(OOD_to_Sonar_fd[0], O_NONBLOCK);
/*   set_fl(OOD_to_Sonar_fd[1], O_NONBLOCK); */
   set_fl(Sonar_to_OOD_fd[0], O_NONBLOCK);
/*   set_fl(Sonar_to_OOD_fd[1], O_NONBLOCK); */
   set_fl(Sonar_telemetry_fd[0], O_NONBLOCK);
/*   set_fl(Sonar_telemetry_fd[1], O_NONBLOCK); */

/*  uncomment when Engineer Module created
   if ((Engineer = fork()) == 0)
   {
       printf("ENGINEER Module forked \n");
       Eng_mod();
   }

 */
   if ((Navigator1 = fork()) == 0)
   {
       printf("NAVIGATOR1 Module forked \n");
       Nav1_mod();
   }


   if ((Navigator3 = fork()) == 0)
   {
       printf("NAVIGATOR3 Module forked \n");
       Nav3_mod();
   }


   if ((Sonar = fork()) == 0)
   {
       printf("SONAR Module forked \n");
       Sonar_mod();
   }

}
```

```
/*************************************************************************/
/*zero_pipe_streams()    Zero pipe prior to write ensures only           */
/*                       1 message on pipe (prevents buffer overflow)     */
/*************************************************************************/

void zero_pipe_streams()
{
/*   uncomment when Engineer Module created
    read(Eng_telemetry_fd[0],Read_to_clear,MAXBUFFERSIZE);
*/
    read(Nav1_telemetry_fd[0],Read_to_clear,MAXBUFFERSIZE);
    read(Nav3_telemetry_fd[0],Read_to_clear,MAXBUFFERSIZE);
    read(Sonar_telemetry_fd[0],Read_to_clear,MAXBUFFERSIZE);
}

/*************************************************************************/
/*send_telemetry_vector sends each child process a copy of current parameters */
/*************************************************************************/

void send_telemetry_vector(buffer_received)
char * buffer_received;
{
   zero_pipe_streams();    /* Ensures only current telemetry in buffer      */
   if (TRACE) printf("Send telemetry vector: %s\n",buffer_received);
/*  uncomment when Engineer Module created
   write(Eng_telemetry_fd[1],buffer_received,MAXBUFFERSIZE);
*/
   write(Nav1_telemetry_fd[1],buffer_received,MAXBUFFERSIZE);
   write(Nav3_telemetry_fd[1],buffer_received,MAXBUFFERSIZE);
   write(Sonar_telemetry_fd[1],buffer_received,MAXBUFFERSIZE);
   strcpy(buffer_received,"");
}

/*************************************************************************/
/* send_command sends messages to all child processes (Does not zero buffer)  */
/*************************************************************************/

void send_command(buffer_received)
char * buffer_received;
{

    if ((strcmp(command,"") != 0) && (strcmp(command,"#") != 0))
    {
        if (TRACE)
            printf("Send_command module: Command sent %s\n",buffer_received);
/*                  uncomment when Engineer Module created
        write(OOD_to_Eng_fd[1],buffer_received,MAXBUFFERSIZE);
*/
        write(OOD_to_Nav1_fd[1],buffer_received,MAXBUFFERSIZE);
        write(OOD_to_Nav3_fd[1],buffer_received,MAXBUFFERSIZE);
        write(OOD_to_Sonar_fd[1],buffer_received,MAXBUFFERSIZE);

        strcpy(buffer_received,"");
        if (TRACE) printf("Send_command module complete.\n");
    }
}
```

```
/*****************************************************************************/
/*       read_pipe_streams checks for messages from child processes          */
/*****************************************************************************/

void read_pipe_streams()  /* Only if all are processed                       */
{
    if (TRACE) printf("begin read_pipe_streams()\n");

    if ((strcmp(Navigator1_string_read,"") == 0) &&
        (strcmp(Navigator3_string_read,"") == 0) &&
        (strcmp(Sonar_string_read,"") == 0) )
    {
       if (TRACE) printf("read_pipe_streams()\n");

/*              uncomment when Engineer Module created
       read(Eng_to_OOD_fd[0],Engineer_string_read,MAXBUFFERSIZE);
*/
       if (TRACE) printf("read(Nav1_to_OOD_fd[0]\n");
       read(Nav1_to_OOD_fd[0],Navigator1_string_read,MAXBUFFERSIZE);
       if (TRACE && (strcmp(Navigator1_string_read,"") != 0))
           printf("Main Module: Navigator1 string returned = %s\n",
              Navigator1_string_read);
       if (TRACE) printf("read(Nav3_to_OOD_fd[0]\n");
       read(Nav3_to_OOD_fd[0],Navigator3_string_read,MAXBUFFERSIZE);
       if (TRACE && (strcmp(Navigator3_string_read,"") != 0))
           printf("Main Module: Navigator3 string returned = %s\n",
               Navigator3_string_read);
       if (TRACE) printf("read(Sonar_to_OOD_fd[0]\n");
       read(Sonar_to_OOD_fd[0],Sonar_string_read,MAXBUFFERSIZE);
       if (TRACE && (strcmp(Sonar_string_read,"") != 0))
           printf("Main Module: Sonar string returned = %s\n",Sonar_string_read);
   }
   if (TRACE) printf("completed read_pipe_streams()\n");
}

/*****************************************************************************/
/*     process_pipe_streams Evaluates messages from child processes          */
/*****************************************************************************/

void process_pipe_streams()
{
    static int i = 0;
    int  read_another_line = TRUE;

/*  uncomment when Engineer Module created
    if (strcmp(Engineer_string_read,"") != 0)
    {
        parse_command(Engineer_string_read);
        strcpy(Engineer_string_read,"");
    }
*/

    if (strcmp(Navigator1_string_read,"") != 0)
    {
        if (TRACE) printf("Navigator1_string_read %s\n",Navigator1_string_read);
        parse_command(Navigator1_string_read);
```

```
if (strcmp(command,"NAV1_INITIALIZED") == 0)
{
   NAV1_INITIALIZED = TRUE;
   printf("NAV1_INITIALIZED\n");
}

else if (strcmp(command,"FIX") == 0)
{
    if ((GPSINPROGRESS == TRUE)  &&
        (variable7 == 1 || variable8 == 1)) /*     DGps or Gps fix    */
    {
        READYFORNEXTCOMMAND = FALSE;
        GPSINPROGRESS = FALSE;

        sprintf(command_sent,  "FIX %lf %lf\n",variable1,variable2);
        write_to_execution_level_socket  ();
        strcpy (command_sent, "");
        read_from_execution_level_socket ();

        strcpy (command_sent, "GPS-FIX-COMPLETE");
        write_to_execution_level_socket  ();
        strcpy (command_sent, "");
        read_from_execution_level_socket ();
    }

    else if (variable6 == 2 && GPSINPROGRESS == FALSE)
    /*              LOST TRACK NEED TO GET A FIX                         */
    {
        READYFORNEXTCOMMAND = FALSE;
        GPSINPROGRESS = TRUE;

        printf("Lost Track Obtaining G P S fix\n");
            time_for_next_command = t + 200;
        tcommand = t;
        strcpy (command_sent, "GPS-FIX");
        write_to_execution_level_socket  ();
        strcpy (command_sent, "");
        read_from_execution_level_socket ();
    }

    else if ((variable6 == 1) && (variable9 == 0))
    {
       if (TRACE)printf("%d tactical sent fix = %4.1lf\n",variable7,t);

      sprintf(command_sent,  "FIX %lf %lf\n",variable1,variable2);
       write_to_execution_level_socket  ();
       strcpy (command_sent, "");
       read_from_execution_level_socket ();

       parse_command(Navigator1_string_read);

       sprintf (command_sent,"OCEANCURRENT %lf %lf",variable4,variable5);
       if (TRACE) printf("oceancurrents %s\n",command_sent);
       write_to_execution_level_socket  ();
       strcpy (command_sent, "");
       read_from_execution_level_socket ();

    }
```

```
            strcpy (command_sent, "");
            strcpy (command, "");
        }
        strcpy(Navigator1_string_read,"");
    }

    else if (strcmp(Navigator3_string_read,"") != 0)
    {
        if (TRACE) printf("Navigator3_string_read %s\n",Navigator3_string_read);
        parse_command(Navigator3_string_read);

        if (strcmp(command,"NAV3_INITIALIZED") == 0)
        {
           NAV3_INITIALIZED = TRUE;
        }

        else if (strcmp(command,"REPLAN_COMPLETE") == 0)
        {
            SUB_POINTS = TRUE;
        }
        strcpy(Navigator3_string_read,"");
        strcpy (command_sent, "");
        strcpy (command, "");
    }

    else if (strcmp(Sonar_string_read,"") != 0)
    {
        if (TRACE) printf("Sonar_string_read %s\n",Sonar_string_read);
        parse_command(Sonar_string_read);

        if (strcmp(command,"SONAR_INITIALIZED") == 0)
        {
           /*      write(OOD_to_Sonar_fd[1],"SONAR_SEARCH",MAXBUFFERSIZE);     */
             SONAR_INITIALIZED = TRUE;
             printf("SONAR_INITIALIZED\n");
        }

        else if (strcmp(command,"SONAR_SEARCH_COMPLETE") == 0)
        {
             READYFORNEXTCOMMAND = TRUE;
        }

        else if (strcmp(command,"NEW_WORLD") == 0)
        {
             sprintf(command,"NEW_WORLD %s",sonar_file_name);
             if (TRACE) printf("New_world: %s",command);
             if (TACTICALSTANDALONE)
                 send_command(command);
             else
             {
                 write(OOD_to_Nav1_fd[1],command,MAXBUFFERSIZE);
                 write(OOD_to_Nav3_fd[1],command,MAXBUFFERSIZE);
                 write(OOD_to_Sonar_fd[1],command,MAXBUFFERSIZE);
             }
             strcpy(command, "");

        }
        else if (strcmp(command,"ROTATE_SEARCH_COMPLETE") == 0)
        {
            READYFORNEXTCOMMAND = TRUE;
            strcpy (command_sent, "ROTATE 0");
            write_to_execution_level_socket  ();
            strcpy (command_sent, "");              /* a null order */
            read_from_execution_level_socket ();
        }


        else if ((strcmp(command,"SONAR_725")  == 0) ||
                 (strcmp(command,"SONAR_1000") == 0) ||
                 (strcmp(command,"SONAR-725")  == 0) ||
                 (strcmp(command,"SONAR-1000") == 0) ||
                 (strcmp(command,"SONAR725")   == 0) ||
                 (strcmp(command,"SONAR1000")  == 0))
        {
            strcpy (command_sent,Sonar_string_read);
          if (TRACE) printf("%s\n",command_sent);
            write_to_execution_level_socket  ();
            strcpy (command_sent, "");              /* a null order */
            read_from_execution_level_socket ();
        }


        else if ((strcmp(command,"COLLISION_THREAT") == 0)
                    && (COLLISION_IMMENENT == FALSE) == 0)
        {
            COLLISION_IMMENENT = TRUE;
            strcpy (command_sent, "RPM -700");
            write_to_execution_level_socket  ();
            strcpy (command_sent, "");
            read_from_execution_level_socket ();

        }

        strcpy (Sonar_string_read,"");
        strcpy (command_sent, "");
        strcpy (command, "");
    }

}

/****************************************************************************/
/* parse_command breaks apart messages into components and sets variables  */
/****************************************************************************/
void parse_command(char * command_to_parse)
{
   static int  counter = 0;
   int  read_another_line = TRUE;

   strcpy(command_sent,command_to_parse);

   sscanf (command_to_parse, "%s",command);
   uppercase();
```

```
if (strcmp(command,"REPLAN") == 0)
{
    if (TRACE) printf("[parse_command () REPLAN]\n");
    /*SUB_POINTS           = TRUE; */
    FLIP_FLOP            = FALSE;
    READYFORNEXTCOMMAND = FALSE;

    /*fclose (subpoints_file_ptr);*/

    sscanf (command_to_parse, "%s %s",command,sonar_file_name);
    uppercase();
}


 else if (strcmp(command,"EXECUTION_INITIALIZED") == 0)
 {
     EXECUTION_INITIALIZED = TRUE;

     printf("EXECUTION_INITIALIZED\n");

     return;
 }


 else if (strcmp(command,"NEW_WORLD") == 0)
 {
    sscanf (command_to_parse, "%s %s",command,sonar_file_name);
    return;
 }

  else if (strcmp(command,"AUV_STATE") == 0)
 {
     fprintf(telemetry_file_ptr,"%s \n",command_to_parse);

     return;
 }


 else if ( (strcmp(command,"GYRO-ERROR")     == 0) ||
           (strcmp(command,"POSTURE")        == 0) ||
           (strcmp(command,"DIVE-TRACKER1") == 0) ||
           (strcmp(command,"DIVE-TRACKER2") == 0) )

 {
     counter++;
     if (counter > 3)
     {
         INITIALIZING = TRUE;
         if (TRACE) printf("INITIALIZING = TRUE\n");
     }

     return;
 }


 else if ( ( (strcmp(command,"STABLE") == 0) && (GPSINPROGRESS == FALSE) ) )
 {
    if (TRACE) printf("\n[parse_command () STABLE]\n");
if (SUB_POINTS == FALSE)
{
    WAYPOINT_OR_HOVERPOINT = FALSE;
    READYFORNEXTCOMMAND    = TRUE;

    if (TRACE) printf("[parse_command () SUB_POINTS == FALSE]\n");
    if (TRACE) printf("time : %lf %s x: %lf y: %lf z: %lf course: %lf\n",
                   t,command_to_parse,x,y,z,psi);
}

else
{
    if(FLIP_FLOP == TRUE)
    {
        FLIP_FLOP = FALSE;
        tcommand = t;

        sscanf (command_sent2, "%s %lf %lf %lf",
            command,&variable1,&variable2,&variable3);



        if (abs(variable2 - y) + abs(variable1 - x) > 6.0)
        {
            ordered_course =(180/M_PI)*(atan2(variable2-y,variable1-x));
            if (TRACE)printf("ORDERED COURSE : %lf \n",ordered_course);
        }

        sprintf(command_sent,"%s %lf %lf %lf %lf",
                command,variable1,variable2,variable3,ordered_course);
        if (TRACE) printf("FLIP_FLOP %s\n",command_sent);
        write_to_execution_level_socket  ();
        strcpy (command_sent, "");
        read_from_execution_level_socket ();
        return;
    }

    else
    {
        while (fgets (command_sent, 120, subpoints_file_ptr) != NULL)
        {
            sscanf (command_sent, "%s %lf %lf %lf %lf %lf",command,
                 &variable1,&variable2,&variable3,&variable4,&variable5);
            uppercase();

            if (strcmp(command,"SEGMENT") != 0){}

            else
            {
                READYFORNEXTCOMMAND = FALSE;
                /*SUB_POINTS = TRUE;*/
                FLIP_FLOP = TRUE;
                WAYPOINT_OR_HOVERPOINT = TRUE;

                tcommand = t;
```

```
                        if (TRACE) printf("Inside :FLIP_FLOP == FALSE");
                        if (abs(variable2 - y) + abs(variable1 - x) > 6.0)
                        {
                            ordered_course = (180/M_PI)*(atan2 (variable2 - y,
                                                     variable1 - x));
                            if (TRACE) printf("ORDERED COURSE: %lf \n",
                                ordered_course);
                        }

                        sprintf(command_sent,"HOVER %lf %lf %lf",
                           variable1,variable2,ordered_depth,ordered_course);
                        if (TRACE)
                           printf("hover %s\n",command_sent);

                        sprintf(goal,"Point %5.1lf %5.1lf %5.1lf Goal\n",
                           variable4,variable5,ordered_depth);

                        write(OOD_to_Nav3_fd[1],goal,MAXBUFFERSIZE);
                        write_to_execution_level_socket  ();
                        strcpy (command_sent, "");
                        read_from_execution_level_socket ();

                        sprintf(command_sent2,"HOVER %lf %lf %lf",
                           variable4,variable5,ordered_depth);

                      return;
                    }
                }
            }
            READYFORNEXTCOMMAND = TRUE;
            SUB_POINTS = FALSE;
            WAYPOINT_OR_HOVERPOINT = FALSE;

            if (TRACE) printf("subpoint file close 1\n");
            fclose(subpoints_file_ptr);
        }
    }

    else if ((GPSINPROGRESS == TRUE ) &&( (strcmp(command,"STABLE") == 0)   ||
              (t > time_for_next_command) ))
    {
       READYFORNEXTCOMMAND = TRUE;
       GPSINPROGRESS = FALSE;

       time_for_next_command = t + 150.0;
       tcommand = t;
       if (TRACE) printf("Time : %lf %s x: %lf y: %lf z: %lf  course: %lf\n",
              t,command_to_parse,x,y,z,psi);
       if (TRACE) printf("Time for next command = %lf\n",time_for_next_command);
    }
```

```
  else if (strcmp(command,"WAIT") == 0)
    {
       READYFORNEXTCOMMAND = FALSE;
       WAITINPROGRESS = TRUE;

       sscanf (command_to_parse, "%s %lf",command,&variable1);
       uppercase();
       if (TRACE) printf("wait  %lf\n",variable1);
       time_for_next_command = t + variable1;
       tcommand = t;
       if (TRACE) printf("Time for next command  %lf\n",time_for_next_command);
    }

    else
    {
        sscanf (command_to_parse, "%s %lf %lf %lf %lf %lf %d %d %d %d",
           command,&variable1, &variable2, &variable3, &variable4,
                   &variable5, &variable6, &variable7, &variable8, &variable9);

        uppercase();

        if (TRACE) printf("ParseCmd V1 = %lf V2 = %lf v3 = %lf v4 = %lf ",
             variable1, variable2, variable3, variable4);
        if (TRACE) printf(" v5 = %lf v6 = %d v7 = %d v8 = %d v9 = %d\n",
             variable5, variable6, variable7, variable8, variable9);

    }

}


/*******************************************************************************/
/*          main() for use with TACTICALSTANDALONE (No Prolog)                 */
/*******************************************************************************/

#if defined(TACTICAL_STANDALONE)

main()
{
    TACTICALSTANDALONE = TRUE;
    printf("Initialize \n");
    initialize();
    printf("Initialization complete \n");
    READYFORNEXTCOMMAND = TRUE;

    for (;;)
    {
        strcpy(command_sent,"");
        strcpy(command,"");
        ood_command_loop();
    }
}

#endif
```

```
/*****************************************************************************/
/*                              ood(cmd)                                     */
/*****************************************************************************/

int ood(cmd)

char * cmd;

{
    strcpy(command_sent,cmd);
/*  printf("CMD sent : %s\n",cmd);                                           */

    parse_command(cmd);
    strcpy(strategic_command,command);
/*  printf("strategic_command %s\n",strategic_command);                      */

    ood_command_loop();
    return(RETURN_VALUE);
}


/*****************************************************************************/
/*ood_command_loop() sets flags and reads mission file for TACTICALSTANDALONE */
/*****************************************************************************/

void ood_command_loop()
{
    int  read_another_line = TRUE;

    if ( ((WAITINPROGRESS == TRUE) || (GPSINPROGRESS == TRUE ))
           && (t > time_for_next_command) )
    {
        printf("TIME FOR NEXT COMMAND TIMED OUT! Time %lf\n",t);
        READYFORNEXTCOMMAND = TRUE;
        GPSINPROGRESS = FALSE;
        WAITINPROGRESS = FALSE;
    }

    if ( (COLLISION_IMMENENT == TRUE) && (abs(u) < .3))
    {
        COLLISION_IMMENENT = FALSE;
        READYFORNEXTCOMMAND = FALSE;
        time_for_next_command = t + 100000;

        printf("Mission abort Shutting Down\n");
        strcpy (command_sent, "ABORT");
        write_to_execution_level_socket  ();
        strcpy (command_sent, "");
        read_from_execution_level_socket ();
        sleep(90);
    }

/*
   Below section is for determining when first SUB_POINT should be sent and
   opening up the SUB_POINTS file
*/
```

```
if ((SUB_POINTS == TRUE) && (WAYPOINT_OR_HOVERPOINT == TRUE) &&
    (OKTOSENDHOVERPOINT == TRUE))
{
    OKTOSENDHOVERPOINT = FALSE;

    if (TRACE) printf("OPENING SUBPOINTS FILE \n");
    if ((subpoints_file_ptr = fopen (SUB_POINTS_FILE, "r"))
         == ((FILE *) 0))
    {
        printf("INVALID SUB_POINTS FILE NAME  %s\n",SUB_POINTS_FILE);
        fclose (subpoints_file_ptr);
    }

    else
    {
        /*SUB_POINTS = TRUE;*/
        FLIP_FLOP = FALSE;
        READYFORNEXTCOMMAND = FALSE;
        read_another_line = TRUE;

        while (read_another_line == TRUE)
        {
            if (fgets (command_sent, 120, subpoints_file_ptr) != NULL)
            {
                sscanf (command_sent, "%s",command);
                uppercase();

                if (strcmp(command,"SEGMENT") != 0)
                read_another_line = TRUE;

                else
                {
                    tcommand = t;
                    read_another_line = FALSE;
                    parse_command(command_sent);
                    if (abs(variable5 - y) + abs(variable4 - x) > 6.0)
                    {
                        ordered_course = atan2 (variable5 - y,
                                                variable4 - x);
                        ordered_course = (180/M_PI)*(ordered_course);
                    }

                    /* are we stable ? need to wait until STABLE reported */

                    sprintf(command_sent,"HOVER %lf %lf %lf %lf",
                    variable4,variable5,ordered_depth,ordered_course);
                    if (TRACE) printf("command sent %s\n",command_sent);
                    write_to_execution_level_socket  ();
                    strcpy (command_sent, "");
                    read_from_execution_level_socket ();
                    strcpy(Navigator3_string_read,"");

                    return;
                }
            }
        }
    }
}
```

```
    if ((READYFORNEXTCOMMAND == TRUE) || ((t - tcommand) > 600.0))
    {
        READYFORNEXTCOMMAND = TRUE;
        if ((TACTICALSTANDALONE == TRUE) && (INITIALIZING == FALSE))
        {
            read_from_mission_file ();
            send_command(command_sent);
        }
    }

    system_time                = time (NULL);
    system_tmp                 = localtime (&system_time);
    string_compare();
}


/*****************************************************************************/
/* string_compare()       Commencing string compare section                   */
/*****************************************************************************/

int string_compare()
{

    strcpy(command_sent,cmd);
    strcpy(strategic_command,command);

    if(strcmp(strategic_command,"INITIALIZE") == 0)
    {
       initialize();

       RETURN_VALUE = 1;
    }

    else if(strcmp(strategic_command,"WAYPOINT") == 0)
    {
        WAYPOINT_OR_HOVERPOINT = TRUE;
        OKTOSENDHOVERPOINT     = TRUE;
        READYFORNEXTCOMMAND    = FALSE;
        /*SUB_POINTS              = TRUE;*/
        tcommand = t;
        ordered_depth  = variable3;

        if (TRACE) printf("Waypoint command x %lf y %lf z %lf\n",
                  variable1,variable2,variable3);

        sprintf(goal,"Point %lf %lf %lf Goal\n",  /*     Next waypoint         */
                variable1,variable2,variable3);

        if (TACTICALSTANDALONE)
            send_command(goal);
        else
        {
            write(OOD_to_Nav1_fd[1],goal,MAXBUFFERSIZE);
            write(OOD_to_Nav3_fd[1],goal,MAXBUFFERSIZE);
            write(OOD_to_Sonar_fd[1],goal,MAXBUFFERSIZE);
        }
        strcpy (command, "REPLAN");

        if (TACTICALSTANDALONE)
            send_command(command);
        else
        {
            write(OOD_to_Nav1_fd[1],command,MAXBUFFERSIZE);
            write(OOD_to_Nav3_fd[1],command,MAXBUFFERSIZE);
            write(OOD_to_Sonar_fd[1],command,MAXBUFFERSIZE);
        }

        if (TRACE) printf("WAYPOINT return 1 \n");
        RETURN_VALUE = 1;
    }


    else if(strcmp(strategic_command,"HOVER") == 0)
    {
        WAYPOINT_OR_HOVERPOINT = TRUE;
        OKTOSENDHOVERPOINT     = TRUE;
        READYFORNEXTCOMMAND    = FALSE;

        if (TRACE) printf("HOVER command x %lf y %lf z %lf course %lf\n",
                          variable1,variable2,variable3,variable4);

        ordered_depth  = variable3;
        ordered_course = variable4;

        sprintf(goal,"Point %lf %lf %lf Goal\n",
                variable1,variable2,variable3);


        if (TACTICALSTANDALONE)
           send_command(goal);
        else
        {
            write(OOD_to_Nav1_fd[1],goal,MAXBUFFERSIZE);
            write(OOD_to_Nav3_fd[1],goal,MAXBUFFERSIZE);
            write(OOD_to_Sonar_fd[1],goal,MAXBUFFERSIZE);
        }

        strcpy (command, "REPLAN");

        if (TACTICALSTANDALONE)
            send_command(command);
        else
        {
            write(OOD_to_Nav1_fd[1],command,MAXBUFFERSIZE);
            write(OOD_to_Nav3_fd[1],command,MAXBUFFERSIZE);
            write(OOD_to_Sonar_fd[1],command,MAXBUFFERSIZE);
        }
            if (TRACE) printf("HOVER return 1 \n");
            RETURN_VALUE = 1;
    }
```

```
else if(strcmp(strategic_command,"COURSE") == 0)
{
    if (WAYPOINT_OR_HOVERPOINT == TRUE) READYFORNEXTCOMMAND = FALSE;
    tcommand = t;

    if (TRACE) printf("Course %lf\n",variable1);
    sprintf(command_sent,"COURSE %lf",variable1);
    write_to_execution_level_socket  ();
    strcpy (command_sent, "");
    read_from_execution_level_socket ();

    if (TRACE) printf("COURSE return 1 \n");
    RETURN_VALUE = 1;
}


else if(strcmp(strategic_command,"DEPTH") == 0)
{
    if (WAYPOINT_OR_HOVERPOINT == TRUE) READYFORNEXTCOMMAND = FALSE;
    tcommand = t;
    ordered_depth = variable1;

    sprintf(command_sent,"%s %lf\n",command,variable1);
    if (TRACE) printf("command_sent after sprintf %s\n",command_sent);
    write_to_execution_level_socket  ();
    strcpy (command_sent, "");
    read_from_execution_level_socket ();

    if (TRACE) printf("DEPTH return 1 \n");
    RETURN_VALUE = 1;
}

else if(strcmp(strategic_command,"SURFACE") == 0)
{
    tcommand = t;

    strcpy (command_sent, "DEPTH 0");
    if (TRACE) printf("depth %lf\n",variable1);
    write_to_execution_level_socket  ();
    strcpy (command_sent, "");
    read_from_execution_level_socket ();

    RETURN_VALUE = 1;
}

else if(strcmp(strategic_command,"STANDOFF") == 0)
{
    if (WAYPOINT_OR_HOVERPOINT == TRUE) READYFORNEXTCOMMAND = FALSE;
    tcommand = t;

    sprintf(command_sent,"%s %lf\n",command,variable1);
    if (TRACE) printf("standoff %lf\n",variable1);
    write_to_execution_level_socket  ();
    strcpy (command_sent, "");
    read_from_execution_level_socket ();

    RETURN_VALUE = 1;
}
```

```
else if(strcmp(strategic_command,"MISSION_COMPLETE") == 0 ||
             strcmp(command,"QUIT") == 0)
{
    READYFORNEXTCOMMAND = FALSE;
    tcommand = t;

    printf("Mission Complete Shutting Down\n");
    strcpy (command_sent, "QUIT");
    write_to_execution_level_socket  ();
    strcpy (command_sent, "");
    read_from_execution_level_socket ();

    sleep (10);
    kill_pipes();
    exit (0);
    RETURN_VALUE = 1;
}

else if(strcmp(strategic_command,"START_TIMER") == 0)
{
  tcommand = t;
  time_out = t + variable1;

  if (TRUE) printf("START TIMER: %s\n",command_sent);
  if (TRUE) printf("TIME OUT %d\n",time_out);

  if (TRACE) printf("START_TIMER return 1 \n");
    RETURN_VALUE = 1;
}


else if(strcmp(strategic_command,"SONAR_SEARCH") == 0)
{
    READYFORNEXTCOMMAND = FALSE;
    tcommand = t;

    if (TACTICALSTANDALONE)
       send_command(strategic_command);

    else
    {
        write(OOD_to_Nav1_fd[1],strategic_command,MAXBUFFERSIZE);
        write(OOD_to_Nav3_fd[1],strategic_command,MAXBUFFERSIZE);
        write(OOD_to_Sonar_fd[1],strategic_command,MAXBUFFERSIZE);
    }

    printf("Commencing sonar search\n");
    strcpy (command_sent, "");
    write_to_execution_level_socket  ();
    read_from_execution_level_socket ();
    strcpy (strategic_command, "");

    RETURN_VALUE = 1;
}
```

```
else if(strcmp(strategic_command,"ROTATE_SEARCH") == 0)
{
    READYFORNEXTCOMMAND = FALSE;
    tcommand = t;
    ordered_course = psi;

    if (TACTICALSTANDALONE)
        send_command(strategic_command);

    else
    {
        write(OOD_to_Nav1_fd[1],strategic_command,MAXBUFFERSIZE);
        write(OOD_to_Nav3_fd[1],strategic_command,MAXBUFFERSIZE);
        write(OOD_to_Sonar_fd[1],strategic_command,MAXBUFFERSIZE);
    }

    printf("Commencing rotate search TIME: %lf\n",tcommand);
    strcpy (command_sent, "rotate 5");
    write_to_execution_level_socket  ();
    strcpy (command_sent, "");
    read_from_execution_level_socket ();
    strcpy (strategic_command, "");
    RETURN_VALUE = 1;
}

else if(strcmp(strategic_command,"ABORT") == 0)
{
    READYFORNEXTCOMMAND = FALSE;
    time_for_next_command = t + 100000;

    printf("Mission abort Shutting Down\n");
    strcpy (command_sent, "ABORT");
    write_to_execution_level_socket  ();
    strcpy (command_sent, "");
    read_from_execution_level_socket ();
    sleep(90);
    kill_pipes();
    exit (0);
    RETURN_VALUE = 1;
}

else if((strcmp(command,"GPS-FIX")      == 0) ||
        (strcmp(command,"GET-GPS-FIX") == 0) ||
        (strcmp(command,"GET_GPS_FIX") == 0))
{
    READYFORNEXTCOMMAND = FALSE;
    GPSINPROGRESS = TRUE;
    tcommand = t;
    time_for_next_command = t + 200;

    printf("Obtaining G P S fix\n");
    strcpy (command_sent, "GPS-FIX");
    write_to_execution_level_socket ();
    strcpy (command_sent, "");
    read_from_execution_level_socket ();

    RETURN_VALUE = 1;
}
```

```
else if(strcmp(strategic_command,"ASK_INITIALIZED") == 0)
{
    if ((NAV3_INITIALIZED == FALSE) || (SONAR_INITIALIZED == FALSE) ||
        (NAV1_INITIALIZED == FALSE))
    {
        if (TRACE) printf("INITIALIZED == FALSE \n");
        read_pipe_streams ();
        process_pipe_streams ();
        strcpy (command_sent, "");
        write_to_execution_level_socket  ();
        read_from_execution_level_socket ();

        if ( (NAV3_INITIALIZED == TRUE) && (SONAR_INITIALIZED == TRUE) &&
             (NAV1_INITIALIZED == TRUE) )
        {
             INITIALIZED = TRUE;

             if (TRACE) printf("INITIALIZED == TRUE \n");

             RETURN_VALUE = 1;
        }

        else
        {
             RETURN_VALUE = 0;
        }
    }

    else
    {
        if (TRACE) printf("INITIALIZED == TRUE \n");

        RETURN_VALUE = 1;
    }
}

else if(strcmp(strategic_command,"ASK_TIME_OUT") == 0)
{
    if ( t > time_out)
    {
        READYFORNEXTCOMMAND = TRUE;

        strcpy (command_sent, "");
        write_to_execution_level_socket  ();
        read_from_execution_level_socket ();

        RETURN_VALUE = 1;
    }

    else
    {
        strcpy (command_sent, "");
        write_to_execution_level_socket  ();
        read_from_execution_level_socket ();

        RETURN_VALUE = 0;
    }
}
```

```
else if (strcmp(strategic_command,"ASK_DEPTH_REACHED") == 0)
{
    if (TRACE) printf("Z : %lf\n",z);
    if (TRACE)printf("ORDERED_DEPTH %lf\n",ordered_depth);
    if (fabs(z - ordered_depth) < 1.0)
    {
        READYFORNEXTCOMMAND = TRUE;

        strcpy (command_sent, "");
        write_to_execution_level_socket  ();
        read_from_execution_level_socket ();

        RETURN_VALUE = 1;
    }

    else
    {
        strcpy (command_sent, "");
        write_to_execution_level_socket  ();
        read_from_execution_level_socket ();

        RETURN_VALUE = 0;
    }
}

else if(strcmp(strategic_command,"ASK_SURFACE_REACHED") == 0)
{
    if ( z < 1)
    {
        strcpy (command_sent, "");
        write_to_execution_level_socket  ();
        read_from_execution_level_socket ();

        RETURN_VALUE = 1;
    }

    else
    {
        strcpy (command_sent, "");
        write_to_execution_level_socket  ();
        read_from_execution_level_socket ();

        RETURN_VALUE = 0;
    }
}

else if(strcmp(strategic_command,"ASK_WAYPT_REACHED") == 0)
{
    if (READYFORNEXTCOMMAND == TRUE)
    {
        strcpy (command_sent, "");
        write_to_execution_level_socket  ();
        read_from_execution_level_socket ();

        if (TRACE) printf("ASK_WAYPT_REACHED return 1 \n");
        RETURN_VALUE = 1;
    }
    else
    {
        strcpy (command_sent, "");
        write_to_execution_level_socket  ();
        read_from_execution_level_socket ();

        RETURN_VALUE = 0;
    }
}


else if(strcmp(strategic_command,"ASK_HOVERPT_REACHED") == 0)
{
    if (READYFORNEXTCOMMAND == TRUE)
    {
        strcpy (command_sent, "");
        write_to_execution_level_socket  ();
        read_from_execution_level_socket ();

        if (TRACE) printf("ASK_HOVERPT_REACHED return 1 \n");
        RETURN_VALUE = 1;
    }

    else
    {
        strcpy (command_sent, "");
        write_to_execution_level_socket  ();
        read_from_execution_level_socket ();

        RETURN_VALUE = 0;
    }
}

else if(strcmp(strategic_command,"ASK_SONAR_SEARCH_COMPLETE") == 0)
{
    if (READYFORNEXTCOMMAND == TRUE)
    {
        strcpy (command_sent, "");
        write_to_execution_level_socket  ();
        read_from_execution_level_socket ();

        RETURN_VALUE = 1;
    }

    else
    {
        strcpy (command_sent, "");
        write_to_execution_level_socket  ();
        read_from_execution_level_socket ();

        RETURN_VALUE = 0;
    }
}
```

```
    else if(strcmp(strategic_command,"ASK_ROTATE_SEARCH_COMPLETE") == 0)
    {
        if (READYFORNEXTCOMMAND == TRUE)
        {
            strcpy (command_sent, "");
            write_to_execution_level_socket  ();
            read_from_execution_level_socket ();

            RETURN_VALUE = 1;
        }

        else
        {
            strcpy (command_sent, "");
            write_to_execution_level_socket  ();
            read_from_execution_level_socket ();

            RETURN_VALUE = 0;
        }
    }


    else if(strcmp(strategic_command,"ASK_GET_GPS_FIX") == 0)
    {
        if (READYFORNEXTCOMMAND == TRUE)
        {
            strcpy (command_sent, "");
            write_to_execution_level_socket  ();
            read_from_execution_level_socket ();

            RETURN_VALUE = 1;
        }

        else
        {
            strcpy (command_sent, "");
            write_to_execution_level_socket  ();
            read_from_execution_level_socket ();

            RETURN_VALUE = 0;
        }
    }

#if defined(TACTICAL_STANDALONE)

     else
     {
         write_to_execution_level_socket ();
         strcpy (command_sent, "");
         strcpy (cmd, "");
         read_from_execution_level_socket ();

         RETURN_VALUE = 0;
     }
```

```
#else

    else
    {
        printf("GARBLED COMMAND SENT %s\n",strategic_command);

        RETURN_VALUE = 0;
    }

#endif

     read_pipe_streams();
     process_pipe_streams();

     strcpy (cmd, "");

     return(RETURN_VALUE);
}


void read_from_mission_file ()
{
   int  parameters_read;
   int  read_another_line = TRUE;

   strcpy(command_sent,"");

   while (read_another_line == TRUE)
   {
       if (fgets (command_sent, 120, script_file_ptr) != NULL)
       {
           parameters_read = sscanf (command_sent, "%s",command);
           bytes_received = strlen (command_sent);

           if      ((parameters_read != 1)              ||
                    (strlen (command)            == 0)  ||
                    (strcmp (command,"#")    == 0)  ||
                    (strlen (command_sent)       == 0)  ||
                    (command_sent [0]            == '\n'))          /* blank line */
              read_another_line = TRUE;

           else
           {
               read_another_line = FALSE;
               tcommand = t;
               time_for_next_command = t + 100;

               if (TRACE) printf ("Time of command %lf\n", tcommand);
               uppercase();
               strcpy(cmd, command_sent);
               parse_command(command_sent);
           }
       }
    else
       {
          fclose(script_file_ptr);
          kill_pipes();
          strcpy (command_sent, "QUIT");
          if (TRACE) printf ("%s\n",command_sent);
          write_to_execution_level_socket  ();
          strcpy (command_sent, "");
          kill_pipes();
       }
   }   /* end while */
   return;
}


void uppercase()
{
    int    index;
    int    number_values = 0;

    if (TACTICALSTANDALONE == TRUE)
        number_values = sscanf (command_sent, "%s", command);
    else
        number_values = sscanf (cmd, "%s", command);

    for (index = 0; index <= (int) strlen (command); index++)
        command [index] = toupper (command [index]);
}
```

The below file contains the initialization script for use by the Tactical level when it is running with the Prolog code. This file is read in when the Strategic level tells the Tactical level to initialize.

```
# INITIALIZATION SCRIPT FOR TEST TANK
# MINI-MOSS-LANDING MISSION

# Divetracker locations
DIVE-TRACKER1    0   0 2
DIVE-TRACKER2  30.8 12.5 2

# Gyro erro (offset from true North)
GYRO-ERROR 0

# Starting position
POSTURE 20.4 8.2 .2 0 0 22

thrusters-on

# Hover until initialization complete
HEADING 22
```

APPENDIX B. SMOOTH MOTION PLANNING SOURCE CODE

The programs listed below are part of the Replanner department. The files included are circtest.c, replanner.c, circle.c. Another file, c_search.c, used by the Replanner department available at *http:/www.cs.nps.navy.mil/research/auv* was not modified by the author and is therefore not included.

```
/*****************************************************************************
*                                                                            *
*   Filename:    circtest.c   circle_test                                   *
*                                                                            *
*   Purpose:     Test program to evaluate circle world (circle.c & c_search.c) *
*                functionality for circle_world robotics project.           *
*                                                                            *
*   Reference:   Advanced Robotics class notes, Dr. Yutaka Kanayama         *
*                                                                            *
*   Author:      Don Brutzman  Brad Leonhardt                                 *
 *                                                                           *
*                                                                            *
*   Date:        18 March 96                                                 *
*                                                                            *
*   Language:    ANSI C                                                      *
*                                                                            *
*   Compile:     cc -g -o circleworld circtest.c -lm                         *
*                                                                            *
*   Execution:   tactical                                                    *
*                                                                            *
*   Comments:    Circle_world is a set of routines for mobile robot modeling *
*                   and two-dimensional path planning.                       *
*                Circle_search performs minimum cost path circle search using *
*                   Dijkstra's algorithm.                                    *
*                Circle_test allows entering circle_world data, computing    *
*                   external and cross-tangents, checking visiblility, and   *
*                   determining a least-cost path from start to goal.        *
*                                                                            *
*                This version is for use on-board Phoenix and is therefore a *
*                     a reduced version which also includes smooth motion.   *
                                                                         *
*                                                                            *
*                All obstacles are modeled as circles.                       *
*                                                                            *
*   Status:      Near-optimal complexity, shortest path solution complete.   *
*                                                                            *
*                                                                            *
*****************************************************************************/


/* Include the next 3 lines in this order for any circle_world application:   */

#ifndef  CIRCLE.C_INCLUDED
#include "circle.c"
#endif
```

```
#include "c_search.c"
#include "../execution/defines.h"
#include "../execution/statevector.h"

#include <time.h>



/*****************************************************************************/
/* function prototypes                                                       */
/*****************************************************************************/
void               replanning                                   ();
void               replanner                                    ();

/* from replanner.c                                                          */
extern void        main_replanner                               ();

/*****************************************************************************/
/* external variables  from tactical.c                                       */
/*****************************************************************************/

extern    int  Nav3_to_OOD_fd[2],OOD_to_Nav3_fd[2];
extern        int  Nav3_telemetry_fd[2];
extern        int  TACTICALPARSE;
extern    char  sonar_file_name[MAXBUFFERSIZE];
extern    char  command[MAXBUFFERSIZE];

/*****************************************************************************/
/* replanning () function called from tactical.c infinite loop               */
/*****************************************************************************/

void replanning ()
{
   static char  filename[40];
   char Nav3_back[MAXBUFFERSIZE];
   static   char goal[MAXBUFFERSIZE];
   char String_read[MAXBUFFERSIZE];
   FILE * in_file_ptr;
   FILE * out_file_ptr;
   TACTICALPARSE = 1;

   for(;;)
   {
      strcpy(String_read,"");
      strcpy(Nav3_back,"");

      if (read(Nav3_telemetry_fd[0],String_read,MAXBUFFERSIZE)== -1){}
             /* Do nothing if no input                                       */
      else
      {
           parse_telemetry_string(String_read);
      }

      if (read(OOD_to_Nav3_fd[0],String_read,MAXBUFFERSIZE)== -1){}
```

```
else
{
   if (TRACE) printf("NAV3 String_read : %s\n",String_read);
   parse_command(String_read);
   uppercase();
   if (TRACE) printf("NAV3 command : %s\n",command);
   if (strcmp (String_read,"QUIT") == 0)
   {
       printf("Terminating Navigator3 Module \n");
       exit(0);
   }

   else if (strcmp (command,"POINT") == 0)
   {
       strcpy(goal,String_read);
       strcpy(command,"");
   }

   else if (strcmp (command,"INITIALIZE") == 0)
   {
       printf("NAV3 INITIALIZE\n\n\n");
       system ("rm circle_world.input*");
       system ("rm circle.input.auv");
       system ("rm circle.auv circle.graph circle.k");
       strcpy(sonar_file_name,"circle.input");
       strcpy(Nav3_back,"NAV3_INITIALIZED");

       if (TRUE) printf("NAV3 INITIALIZED\n");
       strcpy(command,"");
   }

   else if (strcmp (command,"REPLAN") == 0)
   {
       system ("rm circle_world.input*.auv");
       system ("rm circle.input*.auv");
     strcpy(filename,sonar_file_name);
       if (TRACE)printf("FILENAME %s\n",filename);
       strcat(filename,".auv");
     /*  strcpy(sonar_file_name,"circle.input"); */
       if ((out_file_ptr = fopen (filename, "a")) == ((FILE *) 0))
       {
           printf("INVALID INPUT FILE NAME  %s\n",filename);
           fclose  (out_file_ptr);
       }

       fprintf (out_file_ptr,"Point %f %f %f Start\n",x,y,z);
       fprintf (out_file_ptr,"%s",goal);

       if ((in_file_ptr = fopen (sonar_file_name, "r")) == ((FILE *) 0))
       {
           printf("INVALID INPUT FILE NAME  %s\n",sonar_file_name);
           fclose  (in_file_ptr);
       }

       while ((fgets (String_read, 120, in_file_ptr) != NULL))
           fprintf (out_file_ptr,String_read);
       fclose  (in_file_ptr);
       fclose  (out_file_ptr);
```

```
            if (TRACE) printf("Commencing Replanning on file %s\n",filename);
            replanner(filename);
            if (TRUE) printf("Waypoint Replanning Completed\n");
            strcpy(Nav3_back,"REPLAN_COMPLETE");
            strcpy(command,"");
        }
    }



    if (strcmp (Nav3_back,"") != 0)
    {
        write(Nav3_to_OOD_fd[1],Nav3_back,MAXBUFFERSIZE);
    }
    if (strcmp (Nav3_back,"REPLAN_COMPLETE") == 0 && (TRACE))
        printf("%s\n",Nav3_back);
  }
}

/*************************************************************************/
/* replanner (infilename) called from above if REPLAN command received     */
/*************************************************************************/

void replanner (infilename)

/*  Declarations and initializations:                    */
char * infilename;

{
    double          x, y, r;
    double          xmin, xmax, ymin, ymax;  /* min & max values             */
    char            answer = 'y',
                    title    [80], /* string array for graph title          */
                    command [160], /* string array for system commands      */
                    date     [32], /* string for today's date               */
                    label    [40]; /* string for path label                 */
    time_t          today;

    int             mode1, mode2,  /* rotation directions                   */
                    i, j, k,       /* indices                               */
                    copies,
                    search_type,   /* DIJKSTRA or A_STAR search             */
                    plot_each_leg; /* TRUE or FALSE                         */

    Point           start,     goal;
    Point           point1,    point2,    point3,   point4;
    Segment         segment0,  segment1,  segment2;
    Circle          circle1,   circle2;
    Tangent         tangent1,  tangent2;
    Arc             arc1,      arc2;
    Configuration  *config1,  *config2;
    Path           *path0,    *path1,    *path2;
    Circle_list    *circle_ptr;
    Circle_world   *circle_world;
    /* instantiate default paths using a pseudo-segment; instantiate config's */
    point1.x      = 0.0;
    point1.y      = 0.0;
    segment1      = make_segment (point1, point1);
    path0         = create_path  (segment1);
    path1         = create_path  (segment1);
    path2         = create_path  (segment1);

    config1       = (Configuration *) malloc (sizeof (Configuration));
    config2       = (Configuration *) malloc (sizeof (Configuration));

    circle_world = (Circle_world  *) malloc (sizeof (Circle_world));

    answer = 'y';
    if (answer == 'y' || answer == 'Y')
    {
        retrieve_circle_world (infilename,circle_world);
        if  (circle_world != NULL)
        {
             point1.x = circle_world->start.x;
             point1.y = circle_world->start.y;
             point2.x = circle_world->goal.x;
             point2.y = circle_world->goal.y;
             if (TRACE) printf ("\n\n   Start point = (%5.2f, %5.2f)",
                  point1.x, point1.y);
             if (TRACE) printf ("\n\n   Goal  point = (%5.2f, %5.2f)",
                  point2.x, point2.y);
             if (TRACE) printf ("\n\n   Circles in circle_world:  %d",
                      circle_world->degree);
       }
       else
       {
             if (TRACE) printf ("\nRetrieval of circle_world file
unsuccessful.");
             answer = 'n';
       }
    }
    if (TRACE) printf   ("\nrm %s\n",      GRAPH_FILENAME);
    sprintf (command, "rm %s", GRAPH_FILENAME);
    system  (command);

    if (TRACE) printf   ("rm %s\n",        AUV_FILENAME);
    sprintf (command, "rm %s", AUV_FILENAME);
    system  (command);

    if (TRACE) printf   ("rm %s\n",        K_FILENAME);
    sprintf (command, "rm %s", K_FILENAME);
    system  (command);

/*    if (TRACE) printf   ("rm %s\n",        REPLANNER_FILENAME);
    sprintf (command, "rm %s", REPLANNER_FILENAME);
    system  (command);
*/

    /* create a segment and a path direct from start to goal                  */

    start    = point1;
    goal     = point2;
    segment0 = make_segment (point1, point2);
    if (TRACE) printf("\nStraight-line path from start to goal cost = %4.2f\n",
            segment_cost (segment0));
    path0 = create_path (segment0);
    sprintf (label, "Straight line start to goal (cost = %4.2f)",
             segment_cost (segment0));
    path0->label = label;

/*  Circle world data entry complete.                                        */

/*  Output circle world in .graph point pair format and .auv data format     */
    graph_world  (circle_world, GRAPH_FILENAME);
    output_world (circle_world, AUV_FILENAME);


/*  Determine search type and whether to plot each least cost leg found:    */
    answer = '*';
    while ((answer != 'd') && (answer != 'D') &&
           (answer != 'a') && (answer != 'A'))
    {
       if (answer != '*')
           if (TRACE) printf("\n*Please answer D for Dijkstra;A for A-star\n");
       /*scanf  ("%c", &answer);  hack to clear carriage return from buffer*/
      if (TRACE) printf("Do you want a Dijkstra search or an A-star search?\n");
      /* scanf  ("%c", &answer); (bjl) */
       answer = 'a';
       if (answer == 'd' || answer == 'D') search_type = DIJKSTRA;
       if (answer == 'a' || answer == 'A') search_type = A_STAR;

    }

    /*scanf  ("%c", &answer);  hack to clear carriage return from buffer     */
    if (TRACE) printf("Do you want to plot each least cost leg found?       ");
    /* scanf  ("%c", &answer);(bjl)*/
    plot_each_leg = FALSE;
/*-------------------------------------------------------------------------*/

    circle_search (search_type, plot_each_leg, circle_world, path1, path2);

    answer = 'y';
    if (answer == 'y' || answer == 'Y')
    {
        graph_path  (path1, circle_world, GRAPH_FILENAME);
        output_path (path1, AUV_FILENAME);
    }

    /* square off the graph data    */
    center_graph_window (GRAPH_FILENAME, &xmin, &xmax, &ymin, &ymax,
                         GRAPH_STRETCH);

/*main_replanner();*/
}
```

```
/*******************************************************************************/
/*
 Program:          replanner.c  AUV tactical level Navigator3 replanner program

 Authors:          Brad Leonhardt

 Revised:          18 March 96

 System:           SUN Voyager Solaris 2.4 OS; SGI Irix 5.3
 Compiler:         Sun C; IRIX cc

 Compilation:
     [auvonyx]     auv/brad/ood>> make tactical

                   auv/brad/circle-world>>cc k_world.c -o k_world -lm
 Execution:
     [Irix ]       tactical>> tactical


 Description:      tactical level replanner code for Phoenix AUV


 Active changes:  All of it                                                    */

#include <stdio.h>
#include <math.h>
#include <ctype.h>
#define deg_rad M_PI/180
#define error_allowed .0000001
#define out_file "circle.dat"
#define in_file  "circle.k"
#define Yes 0
#define No  1
#define SUBWAYPT_DIST 2
#define sigma 1          /*   .5 Min value before loop de loops occur          */

/*******************************************************************************/
/*              function prototypes                                             */

void            display                                 ();
void            composition                             ();
void            circ                                    ();
void            k_spiral                                ();
void            line_track                              ();
void            reverse_line_track                      ();
void            auv_replanner                           ();
void            main_replanner                          ();
double          norm                                    ();
double        /* Define global variables */
      transform[3],
      transform2[3],
      r_transform[3],
      point[4],next_line_desired[4],
      length,dist,
      alpha,
      x,y,theta,k,
      start_x,start_y,goal_x ,goal_y,
      xpt1,ypt1,xpt2,ypt2,
      initial_theta,line_theta,
      rev_xpt2,rev_ypt2;


int
        count,rotation,not_reverse = 1;

FILE *out_k_ptr,*in_k_ptr;


void display (double * r_transform,int endpoint)
      /*Sends results to screen or file */
{
    double track_dist = 0;
    static double old_x = -9999.0;
    static double old_y = -9999.0;

    track_dist = sqrt(pow((r_transform[0] - old_x),2)
                    + pow((r_transform[1] - old_y),2) );
    if ((track_dist > SUBWAYPT_DIST || endpoint == Yes) &&
         fabs(old_x - r_transform[0]) > error_allowed &&
         fabs(old_y - r_transform[1]) > error_allowed)
    {

        fprintf (out_k_ptr, " %f %f\n", r_transform[0], r_transform[1]);
        old_x = r_transform[0];
        old_y = r_transform[1];
    }
    else if (endpoint == 3)
    {
       printf("goal x = %f  goal y = %f /n",r_transform[0],r_transform[1]);
       fprintf (out_k_ptr, " %f %f\n", r_transform[0], r_transform[1]);
    }
}
```

```
void composition (double * transform,double * transform2)
        /* take current position and move it     */
{
    r_transform[0] = (transform[0] + transform2[0]*cos(transform[2])
       - transform2[1]*sin(transform[2]));
    r_transform[1] = (transform[1] + transform2[0]*sin(transform[2])
       + transform2[1]*cos(transform[2]));
    r_transform[2] = (transform[2] + transform2[2]);
   if (not_reverse)
       display(r_transform,No);
}


void circ(double delta_s, double degrees)
{
    transform2[0] = (1 - (degrees*degrees)/6+
                       pow(degrees,4)/120)*delta_s;
    transform2[1] = (.5 -(degrees*degrees)/24+
                       pow(alpha,4)/720)*degrees*delta_s;
    transform2[2] = degrees;
}


void k_spiral(double length, double alpha)
{
    double k_s,total_distance;
    double delta_s;                          /* distance to travel each move  */
    if (length <= 0) length *= -1;
    total_distance = 0.0;
    delta_s = length/500.0;
       display(r_transform,Yes);
    while (total_distance < length)          /*  Loop until curve traversed   */
    {
        k_s=(alpha*6/pow(length,3)*(total_distance*(length - total_distance)));
      circ(delta_s,k_s*delta_s);           /*circle approx for given length */
      transform[0] = r_transform[0];
      transform[1] = r_transform[1];
      transform[2] = r_transform[2];
      composition(transform, transform2); /* Update current position       */
      total_distance += delta_s;           /* Keep track of length traveled */
    }
    display(r_transform,Yes);

}
```

```
void line_track(double * ld)  /*   Smooth motion to next line desired (ld)    */
{
    double a,b,c,dk_ds = 0.0,dtheta,delta_d;
    double ds = 1.0/10;
    point[3] = 0;
    a = 3.0/sigma;              /*    weighting factors for each term          */
    b = 3.0/(sigma*sigma);
    c = 1.0/(pow(sigma,3));

    do
    {
        point[0] = r_transform[0];             /*  point is current position  */
      point[1] = r_transform[1];
      point[2] = r_transform[2];
      delta_d = -(point[0] - ld[0])*sin(ld[2])    /*Perp. dist error         */
                + (point[1] - ld[1])*cos(ld[2]);
      dk_ds = -(a*(point[3]) + b*(point[2]-ld[2]) + c*delta_d);
      point[3] += dk_ds * ds;
      dtheta = point[3] * ds;
      circ(ds,dtheta);
      composition(point, transform2);             /* update current position */
    }
    while(fabs(dk_ds) > .1 || fabs(delta_d) >.1); /* Exit condition          */
              /*      These values determine how much error to allow      */
     display(r_transform,Yes);
}


void reverse_line_track(double * ld)  /*  Backtrack to find departure point  */
{                                     /*from current line to final position  */
    double a,b,c,dk_ds = 0.0,dtheta,delta_d;
    double ds = 1.0/10;
    a = 3.0/sigma;
    b = 3.0/(sigma*sigma);
    c = 1.0/(pow(sigma,3));
    point[0] = goal_x;     /*Notice that we "pretend" we're at final positon */
    point[1] = goal_y;
    point[2] = norm(M_PI+initial_theta);   /*    Reverse final direction     */
    point[3] = 0;
    not_reverse = 0;

   do
    {
      delta_d = -(point[0] - ld[0])*sin(ld[2])
                + (point[1] - ld[1])*cos(ld[2]);
      dk_ds = -(a*(point[3]-ld[3]) + b*(point[2]-ld[2]) + c*delta_d);
      point[3] += dk_ds * ds;
      dtheta = point[3] * ds;
      circ(ds,dtheta);
      composition(point, transform2);
      point[0] = r_transform[0];          /*   updating "current position"  */
      point[1] = r_transform[1];
      point[2] = r_transform[2];
    }
    while(fabs(dk_ds) > .1 || fabs(delta_d) >.1);   /*  Exit condition      */
          /*      also used to determine allowable error      */
    rev_xpt2 = r_transform[0];   /*    Store position of our exit point     */
    rev_ypt2 = r_transform[1];
    not_reverse = 1;
}



              /*     creates new waypoints if obstacles are incountered     */
void auv_replanner()
{
    double theta1,theta2,length,radius,arc_theta;
    double ld[4];   /*     Line desired values       */
    double temp_transform[3];
    double final_line_desired[4];

    fscanf (in_k_ptr, "%lf %lf %lf %lf", &start_x, &start_y, &goal_x, &goal_y);
    initial_theta = atan2((goal_y-start_y), (goal_x-start_x));/*initial theta*/
    r_transform[0] = start_x;
    r_transform[1] = start_y;
    r_transform[2] = initial_theta;
    fscanf (in_k_ptr, "%lf %lf %lf %lf", &xpt1, &ypt1, &xpt2, &ypt2);
    line_theta = atan2((ypt2-start_y), (xpt2-start_x));
    ld[0] = xpt1;         /*    Line desired values                        */
    ld[1] = ypt1;         /*    Line desired values                        */
    ld[2] = line_theta;   /*    Line desired values                        */
    ld[3] = 0;            /*    Line desired values                        */
    line_track(ld);       /*    Create smooth motion from start to first line */
    dist = sqrt(pow(ypt2 - r_transform[1],2) + /* Once tracking first line   */
            pow(xpt2 - r_transform[0],2)); /* Calculate dist to next pt  */
    transform[0] = r_transform[0];
    transform[1] = r_transform[1];
    transform[2] = r_transform[2];
    transform2[0] = dist;             /*Advance to next point keeping all */
    transform2[1] = 0;                /*values = 0 execept for travel     */
    transform2[2] = 0;                /*in line direction (distance)      */
    composition(transform,transform2);
    display(r_transform,Yes);

    do
    {
       fscanf (in_k_ptr, "%lf %lf %lf %d %lf %lf", &radius, &theta1, &theta2,
           &rotation, &xpt1, &ypt1);
       if (rotation == -1)
       {
           arc_theta = theta1 - theta2;
           if (arc_theta < 0) arc_theta += 360;   /*     Normalize angle     */
       }
       else
       {
          arc_theta = theta2 - theta1;
          if (arc_theta < 0) arc_theta += 360;   /*     Normalize angle     */
       }
```

```
        length = deg_rad*radius*arc_theta;
        arc_theta *= deg_rad;
        k_spiral(length,rotation*arc_theta);       /*     Rotation cw = -1    */
        fscanf (in_k_ptr, "%lf %lf", &xpt2, &ypt2);
        line_theta = atan2((ypt2 - ypt1), (xpt2 - xpt1));
        next_line_desired[0] = xpt1;
        next_line_desired[1] = ypt1;
        next_line_desired[2] = line_theta;
        next_line_desired[3] = 0;
        line_track(next_line_desired);  /*leave arc for next line
           I choose to line_track to eliminate position errors from k-spiral
           ... k-spiral cannot guarantee a final position */

        if (xpt2 != goal_x || ypt2 != goal_y)
                 /*   composition called if not last point     */
        {
           dist=sqrt(pow(ypt2-r_transform[1],2) + pow(xpt2 - r_transform[0],2));
           transform[0] = r_transform[0];
           transform[1] = r_transform[1];
           transform[2] = r_transform[2];
           transform2[0] = dist;
           transform2[1] = 0;
           transform2[2] = 0;
           composition(transform,transform2);
           display(r_transform,Yes);
        }
     }

     while (xpt2 != goal_x && ypt2 != goal_y);
         /*     Check to see if next position is last position                */

     temp_transform[0] = r_transform[0]; /* need to save where we really are  */
     temp_transform[1] = r_transform[1];
     temp_transform[2] = r_transform[2];

     next_line_desired[0] = goal_x;
     next_line_desired[1] = goal_y;
/*          Note the reversed direction                    */
     next_line_desired[2] =norm(M_PI + line_theta);
/*          Commence reverse motion track                  */

     reverse_line_track(next_line_desired);

     r_transform[0] = temp_transform[0];
     r_transform[1] = temp_transform[1];
     r_transform[2] = temp_transform[2];

/*          Determine dist to final exit point             */
     dist = sqrt(pow(rev_ypt2 - r_transform[1],2)
                + pow(rev_xpt2 - r_transform[0],2));
```

```
    transform[0] = r_transform[0];
    transform[1] = r_transform[1];
    transform[2] = r_transform[2];
    transform2[0] = dist;
    transform2[1] = 0;
    transform2[2] = 0;

    composition(transform,transform2);
     display(r_transform,Yes);

    final_line_desired[0] = goal_x;
    final_line_desired[1] = goal_y;
    final_line_desired[2] = initial_theta;
    final_line_desired[3] = 0;

/*              line track to final point                */
    line_track(final_line_desired);

    r_transform[0] = goal_x;
    r_transform[1] = goal_y;
    printf("goal x = %f  goal y = %f /n",r_transform[0],r_transform[1]);
    display(r_transform,3);
}

double norm(input)
double input;
{
    if (input > M_PI)
       return (input - 2*M_PI);
}

void main_replanner()
{
    if ((in_k_ptr = fopen (in_file, "r")) == ((FILE *) 0))
    {
          printf ("input_path:  file open failure!\n");
          return;
    }
    if ((out_k_ptr = fopen (out_file, "w")) == ((FILE *) 0))
    {
        printf ("output_path:  file open failure!\n");
        return;
    }
    auv_replanner();
    fclose  (in_k_ptr);
    fclose  (out_k_ptr);
}
```

```
/*******************************************************************************
*                                                                              *
*   Filename:   circle.c                                                       *
*                                                                              *
*   Purpose:    Define structures, type definitions and functions for          *
*               circle_world robotics project.                                 *
*                                                                              *
*   Reference:  Advanced Robotics class notes, Dr. Yutaka Kanayama             *
*                                                                              *
*   Author:     Don Brutzman                                                   *
*                                                                              *
*   Date:       26 February 92                                                 *
*                                                                              *
*   Modified:   19 November 95                                                 *
*                                                                              *
*   Modified by:Brad Leonhardt                                                 *
*                                                                              *
*   Comments:   circle_world is a set of routines for mobile robot modeling    *
*                  and two-dimensional path planning.                          *
*                                                                              *
*               All obstacles are modeled as circles.                          *
*               Circles are allowed to be adjacent but not overlapping.        *
*               Adjacent (touching) circles do not prevent robot travel        *
*               along either circle circumference.                             *
*                                                                              *
*   Language:   ANSI C                                                         *
*                                                                              *
*   Compile:    cc -g -c circle.c -lm                                          *
*                                                                              *
*   Graphing:   graph  -b  -g 1  -l "circle_world plot"  <  filename  | lpr -g *
*                                                                              *
*   Status:     Shortest path solution complete.                               *
*                                                                              *
*******************************************************************************/

#include <stdio.h>
#include <math.h>
#include <ctype.h>

#define CIRCLE.C_INCLUDED


/****** Circle_world Global Constants ****************************************/

#define CCW              1
#define PLUS             1
#define RIGHT            1
#define POSITIVE         1

#define CW              -1
#define MINUS           -1
#define LEFT            -1
#define NEGATIVE        -1

#define COLINEAR         0
#define CENTER           0
#define ZERO             0
```

```
#define TRUE              1
#define SUCCESS           1
#define YES               1
#define ON                1

#define FALSE             0
#define FAILURE           0
#define NO                0
#define OFF               0

#define VISIBLE           1
#define NONVISIBLE        0
#define TANGENTIAL       -1

#define FATAL             1
#define NONFATAL          0
#define UNDEFINED        -1

#define PI                M_PI                  /* 3.141592653589793       */

#define TANGENTS_OK       TRUE   /* whether or not tangents which pass adjacent*/
                                 /*   to other circles are considered VISIBLE  */

#define EPSILON           1.0E-6 /* error bound in floating point calculations */

#define ARC_FACTOR        1.00   /* factor that arc lays outside circle plot   */

#define TICK_WIDTH        0.50   /* tick width at segment endpoints along path */

#define GRAPH_STRETCH     1.20   /* expansion factor to open out graph window  */

#define SUBDIVISIONS      360    /* number of points used to graphically       */
                                 /*   represent a circle during file output    */

#define DEFAULT_Z         0.0    /* Default pool depth, legal range 0..8 feet  */
                                 /*   where zero feet = surface                */
                                 /* makes circle.auv output data compatible    */
                                 /*   with NPS_Pool_Preview graphics project.  */

#define GRAPH_FILENAME    "circle.graph"   /* name of graph points output file */

#define AUV_FILENAME      "circle.auv"     /* name of high level output file   */

#define K_FILENAME        "circle.k"       /* name of low level output file    */

#define REPLANNER_FILENAME "circle.dat"    /* name of replanner file           */

#define INPUT_FILE        "circle_world.input" /*circles, start, goal pts      */

#define TRACE             FALSE  /* Enable trace printf statements in circle.c */


static float pooltime  = 0.0;    /* used to put time hacks on output objects   */
```

```
/*** List of circle_world Data Structures ************************************/
/*

     External Data Structures        Data Types and Member Labels
     ________________________        ____________________________

     Point                           double         x, y;

     Segment                         Point          point1, point2;

     Circle                          Point          center;
                                     double         radius;

     Tangent                         Circle         circle;
                                     double         angle;

     Arc                             Circle         circle;
                                     double         angle1, angle2;
                                     int            rotation;

    *Configuration                   Tangent        tangent;
                                     double         orientation;

    *Path                            char          *label;
                                     Segment        initial_segment;
                                     int            degree;
                                     Path_list     *path_list;

    *Circle_world                    Point          start, goal;
                                     int            degree;
                                     Circle_list   *circle_list;

     Internal Data Structures
     ________________________

    *Path_list                       Arc            arc;
                                     Segment        segment;
                                     Path_list     *previous, *next;

    *Circle_list                     Circle         circle;
                                     Circle_list   *previous, *next;

*/
```

```
/*** List of circle_world  Functions *****************************************/
/*

     Functions              Parameters
     _________              __________

     error                  (message, fatal)
     make_point             (x, y)
     make_segment           (point1, point2)
     make_circle            (point, radius);
     make_tangent           (circle, angle)
     make_arc               (circle, angle1, angle2, rotation)
     create_configuration   (tangent, angle, configuration)
    *create_path            (initial_segment)
     create_circle_world    (start, goal, circle_world)

     sign                   (x)
     degrees                (angle)
     normalize_rad          (angle)
     normalize2_rad         (angle)
     precede                (angle1, angle2)
     orientation            (point1, point2)
     distance               (point1, point2)

     angle                  (point1, point2, point3)
     S                      (point1, point2, point3)
     area                   (point1, point2, point3)
     order                  (point1, point2, point3)
     between                (point1, point2, point3)

     circumference_point    (circle, angle)
     intersect              (segment, circle)
     visible                (point1, point2, circle_world)

     circle_tangent         (circle1, circle2, mode1, mode2, *config1, *config2)
     arc_cost               (arc)
     segment_cost           (segment)

     augment_path           (arc, segment, path)
     add_circle_to_world    (circle, circle_world)
     find_circle            (n, circle_world)
     retrieve_circle_world  (filename,circle_world)

     graph_path             (path, circle_world, filename)
     graph_world            (circle_world, filename)
     output_path            (path, filename)
     output_world           (circle_world, filename)
     center_graph_window    (filename, xminptr, xmaxptr, yminptr, ymaxptr,
                             magnification)

   See c_search.c (circle_search) for additional data structures and functions.

*/
```

```
/****** Circle_world Data Structures and Type Definitions ********************/

/*--------------------------------------------------------------------------*/

typedef struct Point_type
{
  double x, y;                  /* cartesian coordinate system on a 2D plane  */
}
  Point;

/*--------------------------------------------------------------------------*/

typedef struct Segment_type
{
  Point point1, point2;         /* Note that a Segment is NOT just a          */
}                               /*   collection of (double) x, y coordinate   */
  Segment;                      /*   pairs, i.e. a segment is a pair of Points*/

/*--------------------------------------------------------------------------*/

typedef struct Circle_type
{
  Point  center;
  double radius;                /* radius zero circles are considered points  */
}
  Circle;

/*--------------------------------------------------------------------------*/

typedef struct Tangent_type     /* all circle_world angles are in RADIANS     */
{
  Circle circle;
  double angle;                 /* angle points to the circle tangent point   */
                                /*     relative to the circle center          */
}
  Tangent;

/*--------------------------------------------------------------------------*/

typedef struct Arc_type
{
  Circle circle;
  double angle1,                /* angle1 points to landing point on circle   */
         angle2;                /* angle2 points to leaving point on circle   */
  int    rotation;              /* direction of rotation is either CW or CCW: */
}                               /* RIGHT(+) landing tangent => CCW(+) rotation*/
  Arc;                          /* LEFT (-) landing tangent => CW (-) rotation*/

/*--------------------------------------------------------------------------*/

typedef struct Configuration_type
{
  Tangent tangent;
  double  orientation;          /* orientation is the tangential orientation, */
}                               /*  i.e. the direction that robot is pointing */
  Configuration;

/*--------------------------------------------------------------------------*/
```

```
typedef struct Path_list_type
{
  Arc       arc;                 /* arc is from landing point to leaving point */
  Segment   segment;             /* segment connects leaving point, either to  */
                                 /*   next circle landing point, or to goal    */
  struct    Path_list_type       /* continue this type with doubly-linked list */
           *previous,            /*   of arc/segment combinations until the    */
           *next;                /*   goal is reached                          */
}
  Path_list;

/*---------------------------------------------------------------------------*/

typedef struct Path_type
{
  char        *label;            /* optional string label for each path        */
  Segment      initial_segment;  /* initial segment leaving start point        */
  int          degree;           /* number of arc/segment pairs in the path    */
  Path_list   *path_list;        /* linked list of arc/segment combinations    */
}
  Path;

/*---------------------------------------------------------------------------*/

typedef struct Circle_list_type
{
  Circle       circle;
  struct       Circle_list_type  /* continue this type with  doubly-linked     */
              *previous,         /*   list of remaining circles                */
              *next;
}
  Circle_list;

/*---------------------------------------------------------------------------*/

typedef struct Circle_world_type
{
  Point        start;            /* starting point                             */
  Point        goal;             /* goal     point                             */
  int          degree;           /* number of circles in this world            */
  Circle_list *circle_list;      /* linked list of circles in world            */
}
  Circle_world;
/*---------------------------------------------------------------------------*/
```

```
/****** circle_world Function Declarations ***********************************/

/*--------------------------------------------------------------------------*/

void   error (message, fatal)        /* internal error-handling diagnostic    */

       char *message;                /* error message to be printed           */
       int   fatal;                  /* exit if FATAL, return otherwise       */
{
  fprintf (stderr, "*** Program error: ");
  perror  (message);
  fprintf (stderr, "\n\n");
  if (fatal == FATAL)
  {
     fprintf (stderr, "FATAL error, program exit.\n");
     exit (FAILURE);
  }
  return;
}
/*--------------------------------------------------------------------------*/

Point make_point (x, y)           /* make a point from x, y coordinates       */

       double x, y;
{
    Point point;

    point.x = x;
    point.y = y;

    return point;
}
/*--------------------------------------------------------------------------*/

Segment  make_segment (point1, point2)      /* make a Segment from two points */

         Point point1, point2;
{
    Segment segment;

    segment.point1 = point1;
    segment.point2 = point2;

    return segment;
}
```

```
/*----------------------------------------------------------------------------*/

Circle  make_circle (point, radius) /* create a circle from point & radius */

         Point  point;
         double radius;
{
    Circle circle;

    circle.center = point;
    circle.radius = radius;

    return circle;
}
/*----------------------------------------------------------------------------*/

Tangent  make_tangent (circle, angle)    /* make tangent from circle & angle  */

         Circle circle;
         double angle;
{
    Tangent tangent;

    tangent.circle = circle;
    tangent.angle  = angle;

    return tangent;
}
/*----------------------------------------------------------------------------*/

Arc   make_arc (circle, angle1, angle2, rotation)

      Circle  circle;                 /* create an arc_type from circle data,  */
      double  angle1, angle2;         /* two angles and a direction of rotation*/
      int     rotation;               /* (CW or CCW)                            */
{
    Arc arc;

    /* initialize values */

    arc.circle   = circle;
    arc.angle1   = angle1;
    arc.angle2   = angle2;
    arc.rotation = rotation;

    return arc;
}
```

```
/*-----------------------------------------------------------------------*/

void create_configuration (tangent, orientation, configuration)

     Tangent          tangent;          /* create a configuration_type pointer */
     double           orientation;      /*   from a tangent and an orientation */
     Configuration   *configuration;    /* resulting configuration output      */

{
    if (configuration == NULL)          /* allocate memory if needed           */
    {
        if (TRACE) printf ("\n*** create_configuration:  allocating memory");
        if ((configuration = (Configuration *) malloc (sizeof (Configuration)))
                                == NULL)
            error ("create_configuration:  memory allocation failure", FATAL);
    }

    /* initialize values */

    configuration->tangent     = tangent;
    configuration->orientation = orientation;

    return;
}
/*-----------------------------------------------------------------------*/

static Path  *create_path (initial_segment)

       Segment initial_segment;      /* begin a path_type linked list, using  */
                                     /* a segment which includes start point  */
{
    static Path *path;

    if ((path = (Path *) malloc (sizeof (Path))) == NULL)
        error ("create_path:  memory allocation failure", FATAL);

    /* initialize values */

    path->label               = " ";
    path->degree              = 0;
    path->initial_segment     = initial_segment;
    path->path_list           = ((Path_list *) 0);

    return path;
}
/*-----------------------------------------------------------------------*/
void create_circle_world (start, goal, world)

     Point           start, goal;    /* create a circle_world using           */
                                     /*   start and goal points only          */
     Circle_world   *world;          /* resulting circle_world output         */
{
    if (world == NULL)               /* allocate memory if needed             */
    {
        if (TRACE) printf ("\n*** create_circle_world:  allocating memory");
        if ((world = (Circle_world *) malloc (sizeof (Circle_world))) == NULL)
            error ("create_circle_world:  memory allocation failure", FATAL);
    }
    /* initialize values */

    world->start       = start;
    world->goal        = goal;
    world->degree      = 0;
    world->circle_list = ((Circle_list *) 0);

    return;
}
/*---------------------------------------------------------------------------*/

int     sign (x)                        /* return sign of x as an integer        */
        double x;
{
    if       (x > 0.0)
      return POSITIVE;
    else if  (x < 0.0)
      return NEGATIVE;
    else
      return ZERO;
}
/*---------------------------------------------------------------------------*/

double degrees (angle)                  /* conversion from radians to degrees    */
       double angle;                    /* note no normalization is performed    */
{
    return angle * 180.0 / PI;
}
/*---------------------------------------------------------------------------*/

double normalize_rad (angle)                /* standard normalization range -PI..PI
*/

       double angle;                    /* angle is any real value in RADIANS    */
{
    double x;
    x = angle;
    while (x >   PI) x = x - PI - PI;
    while (x < - PI) x = x + PI + PI;
    return x;
}
/*---------------------------------------------------------------------------*/

double normalize2_rad (angle)               /* alternate normalization range 0..2*PI
*/

       double angle;                    /* angle is any real value in RADIANS    */
{
    double x;
    x = angle;
    while (x > PI + PI) x = x - PI - PI;
    while (x < 0.0)     x = x + PI + PI;
    return x;
}
```

```
/*---------------------------------------------------------------------------*/

int    precede (angle1, angle2)      /* Boolean function for angle precedence */

       double angle1, angle2;        /* angles are any real angles in degrees */
                                     /* return TRUE if angle1 precedes angle2,*/
                                     /*   return FALSE otherwise             */
{
/* note that the input angles are individually normalized to ensure validity  */

    if (normalize_rad (normalize_rad (angle2) - normalize_rad (angle1)) > 0.0)
       return TRUE;
    else                             /* reference:  equation (7) class notes  */
       return FALSE;
}
/*---------------------------------------------------------------------------*/

double orientation (point1, point2)  /* range -PI .. PI                       */

       Point point1, point2;         /* return normalized angle between points*/
{
    if   ((point1.x == point2.x) && (point1.y == point2.y))
          return 0.0;
    else
          return normalize_rad (atan2 (point2.y - point1.y, point2.x -
point1.x));
}
/*---------------------------------------------------------------------------*/

double distance (point1, point2)     /* euclidean distance between two points */
       Point point1, point2;
{
    double deltax = point2.x - point1.x;
    double deltay = point2.y - point1.y;
    return sqrt (deltax * deltax + deltay * deltay);
}
/*---------------------------------------------------------------------------*/

int angle (point1, point2, point3)   /* return angle between three points     */
                                     /* angle order is point1..point2..point3 */
    Point  point1, point2, point3;

{
    return normalize_rad (orientation (point2, point1) -
                         orientation (point2, point3));
}
/*---------------------------------------------------------------------------*/

double S (point1, point2, point3)    /* calculate S function for three points */
                                     /* reference:  equation (14) class notes */
    Point  point1, point2, point3;

/* CCW triples are positive & CW triples are negative, matching conventions.  */

{
    return 0.5 * ((point2.x - point1.x) * (point3.y - point1.y) -
                  (point3.x - point1.x) * (point2.y - point1.y));
}
```

```
/*----------------------------------------------------------------------------*/

double area (point1, point2, point3) /* calculate area between three points   */
                                     /* reference: Proposition 3.1 class notes*/
    Point  point1, point2, point3;
{
    return fabs (S (point1, point2, point3));
}
/*----------------------------------------------------------------------------*/

int    order (point1, point2, point3)/* determine order of three points       */

       Point  point1, point2, point3;/* relationship between three points is  */
                                     /* clockwise (CW), counterclockwise (CCW)*/
                                     /*   or COLINEAR                         */
                                     /* reference:  equation (15) class notes */
{
    if (fabs (S (point1, point2, point3)) <= EPSILON)
       return COLINEAR;              /* floating point error check, correction*/
    else
       return sign (S (point1, point2, point3));
}
/*----------------------------------------------------------------------------*/

int between (point1, point2, point3) /* test for betweenness of point2        */

    Point  point1, point2, point3;
{
    if ((order (point1, point2, point3) == COLINEAR)     &&
        (point1.x <= point2.x) && (point2.x <= point3.x) &&
        (point1.y <= point2.y) && (point2.y <= point3.y))
        return TRUE;
    else
        return FALSE;
}
/*----------------------------------------------------------------------------*/

Point  circumference_point (circle, angle) /* return coordinates of a point   */
                                           /*   on a circle's circumference   */

       Circle circle;  double angle;
{
    return make_point (circle.center.x + circle.radius * cos (angle),
                       circle.center.y + circle.radius * sin (angle));
}
```

```
/*-----------------------------------------------------------------------*/

int intersect (segment, circle)     /* determine if segment intersects circle */

    Segment       segment;
    Circle        circle;

    /**********************************************************************/
    /* return: TRUE       if any  intersection exists within the circle       */
    /* return: FALSE      if no   intersections exist within the circle       */
    /* return: TANGENTIAL if only intersection is within EPSILON of the       */
    /*                              circle circumference (i.e. tangential),   */
    /*                              TANGENTS_OK is defined as TRUE, & neither */
    /*                              segment endpoint is on the circumference. */
    /**********************************************************************/
{
    /* local variable declarations                                          */
    double        height,             /* height of circle center from segment */
                  theta,              /* orientation angle of segment         */
                  difference;         /* difference between circle.radius and */
                                      /* distance of circle.center to segment */

    /* check special case:  circle center is on the line segment          */
    if (between (segment.point1, circle.center, segment.point2))
       {
        if (TRACE) {printf ("\n*** intersect:  betweenness case  ");
                    printf ("(%3.1f, %3.1f) (%3.1f, %3.1f)",
                    segment.point1.x, segment.point1.y,
                    segment.point2.x, segment.point2.y);}
        if  (circle.radius <= EPSILON)    /* this accounts for "point" circles */
            return TANGENTIAL;
        else
            return TRUE;                  /* intersection occurred             */
       }

    /* determine theta = orientation angle of segment                      */
    theta = orientation (segment.point1, segment.point2);

    if (TRACE) printf ("\n*** intersect:  ");

          /* check case where circle.center is on right hand side of segment */
    if    (fabs (normalize_rad (orientation (segment.point2, circle.center) -
theta))
                <= (PI/2.0))
          {
          difference = distance (segment.point2, circle.center) - circle.radius;
          if (TRACE) printf ("case 1  ");
          if (TRACE) printf ("orientation = %f  ",
                 degrees (orientation (segment.point2, circle.center)));
          if (TRACE) printf (" theta = %f  ", theta);
          if (TRACE) printf (" normalize_rad = %f  ",
             degrees (normalize_rad (orientation (segment.point2,
circle.center))));
          }
    else  /* check case where circle.center is on left hand side of segment   */
        if (fabs (normalize_rad (orientation (segment.point1, circle.center) -
theta))
                  >= (PI/2.0))
           {
           difference = distance (segment.point1, circle.center) - circle.radius;
           if (TRACE) printf ("case 2  ");
           }

    else  /* check case where segment length is zero                           */
        if (distance (segment.point1, segment.point2) == 0.0)
           {
           difference = distance (segment.point1, circle.center) - circle.radius;
           if (TRACE) printf ("case 3  ");
           }

    else  /* circle.center is in a voronoi region defined by segment edge     */
           {
           height     = area (segment.point1, segment.point2, circle.center)
                        * 2 / distance (segment.point1, segment.point2);
           difference = height - circle.radius;
           if (TRACE) printf ("case 4  ");
           }

    /*      Now use 'difference' to test for tangency or intersection.        */

    if (TRACE) {printf ("difference = %f  ", difference);
                printf ("(%3.1f, %3.1f) (%3.1f, %3.1f)",
                         segment.point1.x, segment.point1.y,
                         segment.point2.x, segment.point2.y);
                printf ("  circle (%3.1f, %3.1f) ",
                        circle.center.x, circle.center.y);
               }

    if      (difference < - EPSILON) /* circle.radius is greater than the     */
            return TRUE;              /*   circle's distance from the segment  */
                                      /*   thus intersection is TRUE           */

    else if ((fabs (distance (segment.point1, circle.center) - circle.radius)
                 <= EPSILON) ||
             (fabs (distance (segment.point2, circle.center) - circle.radius)
                 <= EPSILON))
            return FALSE; /* ignore external tangency of segment endpoints,   */
                          /*   because solely trying to determine tangency    */
                          /*   with different circles in circle_world         */

    else if ((fabs (difference) <= EPSILON) && TANGENTS_OK == TRUE)
            return TANGENTIAL;

    else
            return FALSE;             /* circle does not intersect segment     */
}
```

```
/*----------------------------------------------------------------------*/

int visible (point1, point2, circle_world)

    Point          point1, point2;    /* determine whether a direct path is    */
    Circle_world  *circle_world;      /*   visible between two points without  */
                                      /*   crossing any circles in circle_world*/

    /*********************************************************************/
    /* return:    VISIBLE if no   intersections exist with circle_world      */
    /* return: NONVISIBLE if any  intersection exists with circle_world      */
    /* return: TANGENTIAL if only intersection(s) are within EPSILON of      */
    /*                            circle circumference(s), i.e. tangential   */
    /*********************************************************************/
{
    /* local variable declarations                                           */
    int          i,                   /* index                                 */
                 visibility;          /* VISIBLE, NONVISIBLE or TANGENTIAL     */
    Segment      segment;             /* intersection of perpendicular & line  */
    Circle       circle;              /* local variable holding current circle */
    Circle_list *world_ptr;           /* pointer to current circle             */

    visibility = VISIBLE;             /* default initialization                */
    segment    = make_segment (point1, point2);
    world_ptr  = circle_world->circle_list; /* first circle in world           */

    /* Note that intersection with circle_world start & goal is not checked.  */

    for (i=1; i <= circle_world->degree; ++i)
    {
        /* check next circle in circle_world for intersections                 */
        circle = world_ptr->circle;

        if       (intersect (segment, circle) == TANGENTIAL)
                 visibility = TANGENTIAL; /* i.e. close enough to be a tangent*/
                                          /*    continue searching             */
        else if  (intersect (segment, circle) == TRUE)
             {
                if (TRACE) {printf ("\n*** visible complete:  NONVISIBLE\n");}
                visibility = NONVISIBLE;
                return visibility;
             }
        world_ptr = world_ptr->next;
    }

    return visibility;
}
```

```
/*-----------------------------------------------------------------------*/

void circle_tangent (circle1, circle2, mode1, mode2, config1, config2)

     Circle          circle1,  /*  input: Leaving circle where tangent starts */
                     circle2;  /*  input: Landing circle where tangent ends   */
     int             mode1,    /*  input: which side of circle1               */
                     mode2;    /*  input: which side of circle2               */
     Configuration  *config1,  /* output: starting configuration pointer      */
                    *config2;  /* output:   ending configuration pointer      */
{
    double  alpha, theta, delta, angle1, angle2; /* local declarations       */
    Circle  circle3;           /*  input: Leaving circle where tangent starts */
    Circle  circle4;           /*  input: Leaving circle where tangent starts */

    circle3 = circle1;
    circle4 = circle2;

    theta = orientation (circle1.center, circle2.center);

    /* Simplified delta angle equation originated by LT Scott Starsman USN    */
    delta = asin ((mode2 * circle2.radius - mode1 * circle1.radius) /
                   distance (circle2.center, circle1.center));
    alpha = normalize_rad (theta - delta);           /* tangential orientation
*/

    angle1 = normalize_rad (alpha - mode1 * PI / 2); /* leaving angle circle1
*/
    angle2 = normalize_rad (alpha - mode2 * PI / 2); /* landing angle circle2
*/

    if (mode1 == CENTER)
    {
        angle1        = 0.0;
        circle3.radius = 0.0;
    }
    if (mode2 == CENTER)
    {
        angle2        = 0.0;
        circle4.radius = 0.0;
    }

    if (TRACE)
       printf ("\n*** circle_tangents:  theta = %f, delta = %f, alpha = %f, \n",
               degrees (theta), degrees (delta), degrees (alpha));
    if (TRACE) printf ("                              angle1 = %f, angle2 = %f\n",
               degrees (angle1), degrees (angle2));

    config1->tangent     = make_tangent (circle3, angle1);
    config1->orientation = alpha;
    config2->tangent     = make_tangent (circle4, angle2);
    config2->orientation = alpha;

    return;
}
```

```
/*-------------------------------------------------------------------------*/

double arc_cost (arc)                 /* euclidean distance cost function     */

       Arc arc;                       /* rotation direction is included in arc */
{
    double delta_angle;

    if      (arc.rotation == CW)
       delta_angle = normalize_rad (arc.angle1) - normalize_rad (arc.angle2);

    else if (arc.rotation == CCW)
       delta_angle = normalize_rad (arc.angle2) - normalize_rad (arc.angle1);

    else if (arc.rotation == ZERO)
       delta_angle = 0.0;

    else
    {
       delta_angle = normalize_rad (arc.angle1) - normalize_rad (arc.angle2);
       printf ("\nIllegal rotation value (%1d) given to arc_cost function.",
               "  Assumed CLOCKWISE.\n", arc.rotation);
    }

    /* circumference portion = 2 * PI * R * (delta_angle / (2 * PI))          */
    return arc.circle.radius * normalize2_rad (delta_angle);
}
/*-------------------------------------------------------------------------*/

double segment_cost (segment)         /* euclidean distance cost function     */

       Segment segment;
{
    return distance (segment.point1, segment.point2);
}
/*-------------------------------------------------------------------------*/

void   augment_path (arc, segment, path)

       Arc              arc;          /* add arc and segment to path          */
       Segment          segment;      /* the order of adding an arc followed  */
       Path            *path;         /* by a segment is a rigorous requirement*/
{
    Path_list          *path_ptr;     /* index pointer to legs on the path    */
    Path_list          *path_node;    /* local variable to build path leg     */

    if ((path_node = (Path_list *) malloc (sizeof (Path_list))) == NULL)
        error ("augment_path:  memory allocation failure!", FATAL);

    /* initialize values of path_node which will augment current path         */
    path_node->arc      = arc;
    path_node->segment  = segment;
    path_node->next     = ((Path_list *) 0);
    path_node->previous = ((Path_list *) 0);

    if (path->degree == 0)            /* first path in path_list to be added  */
        path->path_list = path_node;
    else
    {
        /* point to first leg of path, then find end of current path_list     */
        path_ptr = path->path_list;
        while (path_ptr->next != ((Path_list *) 0))
               path_ptr = path_ptr->next;

        /* now augment current path with new path leg                          */
        path_node->previous = path_ptr;
        path_ptr ->next     = path_node;
    }
    path->degree++;
    if (TRACE)
       printf("\n*** path->degree = %i, augment_path complete\n", path->degree);
    return;
}
/*---------------------------------------------------------------------------*/

void   add_circle_to_world (circle, circle_world)

       Circle          circle;        /* circle to be added to world           */
       Circle_world   *circle_world;  /* current circle_world                  */
{
    Circle_list       *circle_ptr;    /* index pointer to current circle       */
    Circle_list       *circle_node;   /* local variable to build circle leg    */
    double             separation;    /* used to check & prevent circle overlap*/

    if (TRACE) printf ("\n*** add_circle_to_world start\n");

    if ((circle_node = (Circle_list *) malloc (sizeof (Circle_list))) == NULL)
        error ("add_circle_to_world:  memory allocation failure!", FATAL);


    /* initialize values of circle_node which will be added to circle_world   */
    circle_node->circle            = circle;
    circle_node->next              = ((Circle_list *) 0);
    circle_node->previous          = ((Circle_list *) 0);

    if (circle_world->degree == 0) /* first circle in circle_world to be added*/
        circle_world->circle_list = circle_node;

    else
    {
        /* point to first circle in world, then find end of current circles   */
        circle_ptr = circle_world->circle_list;

        while (circle_ptr->next != ((Circle_list *) 0))
        {
          circle_ptr = circle_ptr->next;
        }
        /* now check that pesky last circle                                    */
        /* now add new circle_node to current circle_list in circle world     */
        circle_node->previous = circle_ptr;
        circle_ptr ->next     = circle_node;
    }
    circle_world->degree++;
    if (TRACE) printf ("\n*** circle_world->degree = %i\n",
                       circle_world->degree);
    if (TRACE) printf ("\n*** add_circle_to_world complete\n");

    return;
}
/*---------------------------------------------------------------------------*/

Circle  find_circle (n, circle_world)

        int             n;            /* get the nth circle from circle_world */
        Circle_world    *circle_world;
{
        int          i, nn;
        Circle       circle0;
        Circle_list *circle_ptr;
        nn = n;

        if (circle_world->degree == 0)
        {
            circle0 = make_circle (circle_world->start, 0.0);
            return circle0;
        }
        while ((nn <= 0) || (nn > circle_world->degree))
        {
           if (TRACE) printf ("\n*** find_circle: there are %d circles in
circle_world.",
                          circle_world->degree);
           if (TRACE) printf ("  Which do you want?  ");
            scanf ("%d", &nn); if (TRACE) printf ("\n");
        }
        /* ready to go; point to first circle in world, then find n_th circle */
        i = 1;
        circle_ptr = circle_world->circle_list;
        while ((i < nn) && (circle_ptr->next != NULL))
        {
              circle_ptr = circle_ptr->next;
              ++i;
        }
        return circle_ptr->circle;
}
/*---------------------------------------------------------------------------*/

void   graph_path (path, circle_world, filename)

       /* Print path data for unix 'graph' use, appended to 'filename'        */

       Path            *path;
       Circle_world    *circle_world;
       char            *filename;
```

```
{
    FILE         *file_ptr;
    double        begin_angle, end_angle, delta_angle, angle,
                  midpoint_x, midpoint_y, x1, y1, x2, y2;
    int           i, j, n, rotation;  /* indices, # arc steps & local variable */
    Point         point;
    Circle        arc_circle;
    Path_list    *path_ptr;           /* index pointer to legs on the path     */

    if (TRACE) printf ("\n*** graph_path start\n");

    if (path == NULL) printf ("\n*** path == NULL, error!\n");

     path_ptr = path->path_list;       /* point to first leg of path           */

    if ((file_ptr = fopen (filename, "a")) == ((FILE *) 0))
    {
        error ("graph_path:  file open failure!", NONFATAL);
        return;
    }
    /* print starting line segment                                           */
    if (TRACE) printf (" %f %f\n",        path->initial_segment.point1.x,
                                          path->initial_segment.point1.y);
    if (TRACE) printf (" %f %f\n\" \"\n", path->initial_segment.point2.x,
                                          path->initial_segment.point2.y);
    fprintf (file_ptr, " %f %f\n",        path->initial_segment.point1.x,
                                          path->initial_segment.point1.y);
    fprintf (file_ptr, " %f %f\n\" \"\n", path->initial_segment.point2.x,
                                          path->initial_segment.point2.y);

    /* Print tick marks perpendicular to endpoint of initial segment  */
    if (((path->initial_segment.point2.x != circle_world->start.x)  ||
         (path->initial_segment.point2.y != circle_world->start.y)) &&
        ((path->initial_segment.point2.x != circle_world->goal.x)   ||
         (path->initial_segment.point2.y != circle_world->goal.y)))
    {
       angle = orientation (path->initial_segment.point1,
                            path->initial_segment.point2) + (PI/2.0);
       x1 = path->initial_segment.point2.x - (TICK_WIDTH / 2.0) * cos (angle);
       y1 = path->initial_segment.point2.y - (TICK_WIDTH / 2.0) * sin (angle);
       x2 = path->initial_segment.point2.x + (TICK_WIDTH / 2.0) * cos (angle);
       y2 = path->initial_segment.point2.y + (TICK_WIDTH / 2.0) * sin (angle);
       fprintf (file_ptr, " %f %f\n %f %f\n\" \"\n", x1, y1, x2, y2);
       if (TRACE) printf ("tickmark at end of initial segment: \n");
       if (TRACE) printf (" %f %f\n %f %f\n\" \"\n", x1, y1, x2, y2);
    }
    /* Print path label adjacent to midpoint of initial_segment              */
    midpoint_x = (path->initial_segment.point1.x +
                  path->initial_segment.point2.x) / 2.0;
    midpoint_y = (path->initial_segment.point1.y +
                  path->initial_segment.point2.y) / 2.0;
    if (TRACE) printf (" %f %f\n", midpoint_x, midpoint_y);
    fprintf (file_ptr, " %f %f\n", midpoint_x, midpoint_y);
```

```
if (TRACE)
{
    if (path->label != NULL)  printf ("\"%s \"\n", path->label);
    else                      printf ("\" \"\n");
}
if (path->label != NULL)      fprintf (file_ptr, "\"%s \"\n", path->label);
else                          fprintf (file_ptr, "\" \"\n");

/* print all succeeding arc / line segment combinations                 */
for (i=1; i <= path->degree; ++i, path_ptr = path_ptr->next)
{
    /* Calculate and plot arc traversal points                          */


    begin_angle = normalize2_rad (path_ptr->arc.angle1);
    end_angle   = normalize2_rad (path_ptr->arc.angle2);
    rotation    =                path_ptr->arc.rotation;

    if       (fabs (begin_angle - end_angle) <= EPSILON)
              delta_angle = 0.0;
    else if ((precede (end_angle, begin_angle) && (rotation == CCW)) ||
              (precede (begin_angle, end_angle) && (rotation == CW)))
              delta_angle = PI + PI - fabs (end_angle - begin_angle);
    else
              delta_angle =     rotation * (end_angle - begin_angle);

    delta_angle = normalize2_rad (delta_angle);
    angle       = begin_angle;
    arc_circle  = path_ptr->arc.circle;

    n = (int)((float)(SUBDIVISIONS) * fabs (delta_angle) / (PI + PI) + .5);
    if   ((delta_angle == 0.0) || (n <= 0) || (rotation == CENTER))
          n = 0;  /* perform only one iteration of loop                 */
    else
    {
       /* Print first point of arc without ARC_FACTOR for continuity    */
       point = circumference_point (arc_circle, begin_angle),
       fprintf (file_ptr, " %f %f\n", point.x, point.y);
       if (TRACE) printf (" %f %f\n", point.x, point.y);
    }

    if (TRACE)
    {
        printf ("\n*** graph_path  n = %d, delta_angle = %f\n",
                 n, degrees(delta_angle));
        printf (  "                     begin_angle = %f, ",
                   degrees (begin_angle));
        printf ("end_angle = %f, rotation = %d\n\n",
                   degrees (end_angle), rotation);
    }

    /* Factor radius to graph arc just outside circle circumference      */
    arc_circle.radius *= ARC_FACTOR;
```

```
        /* calculate and print points for the arc starting from initial angle */
        for (j = 0; j <= n; ++j)
        {
            if (TRACE) printf ("*** graph_path:  j = %d, angle = %f, n = %d \n",
                               j, degrees (angle), n);
            point = circumference_point (arc_circle, angle);

            if  (n != 0)  fprintf (file_ptr, " %f %f\n", point.x, point.y);
            if ((n != 0) && (TRACE)) printf (" %f %f\n", point.x, point.y);
            if  (n != 0) angle += (rotation * delta_angle / (double) n);
            angle = normalize2_rad (angle);
        }
        /* calculate and print points for the segment following the arc      */
        point = path_ptr->segment.point1;
        fprintf (file_ptr, " %f %f\n", point.x, point.y);
        if (TRACE) printf (" %f %f\n", point.x, point.y);

        point = path_ptr->segment.point2;
        fprintf (file_ptr, " %f %f\n\" \"\n", point.x, point.y);
        if (TRACE) printf (" %f %f\n\" \"\n", point.x, point.y);

        /* Print tick mark perpendicular to start point of current segment  */
        if (((path_ptr->segment.point1.x != circle_world->start.x)  ||
             (path_ptr->segment.point1.y != circle_world->start.y)) &&
            ((path_ptr->segment.point1.x != circle_world->goal.x)   ||
             (path_ptr->segment.point1.y != circle_world->goal.y)))
        {
           angle = orientation (path_ptr->segment.point1,
                                path_ptr->segment.point2) + (PI/2.0);
           x1 = path_ptr->segment.point1.x - (TICK_WIDTH / 2.0) * cos (angle);
           y1 = path_ptr->segment.point1.y - (TICK_WIDTH / 2.0) * sin (angle);
           x2 = path_ptr->segment.point1.x + (TICK_WIDTH / 2.0) * cos (angle);
           y2 = path_ptr->segment.point1.y + (TICK_WIDTH / 2.0) * sin (angle);
           fprintf (file_ptr, " %f %f\n %f %f\n\" \"\n", x1, y1, x2, y2);
           if (TRACE) printf ("tickmark at start of current segment: \n");
           if (TRACE) printf (" %f %f\n %f %f\n\" \"\n", x1, y1, x2, y2);
        }
        /* Print tick mark perpendicular to final point of current segment  */
        if (((path_ptr->segment.point2.x != circle_world->start.x)  ||
             (path_ptr->segment.point2.y != circle_world->start.y)) &&
            ((path_ptr->segment.point2.x != circle_world->goal.x)   ||
             (path_ptr->segment.point2.y != circle_world->goal.y)))
        {
           angle = orientation (path_ptr->segment.point1,
                                path_ptr->segment.point2) + (PI/2.0);
           x1 = path_ptr->segment.point2.x - (TICK_WIDTH / 2.0) * cos (angle);
           y1 = path_ptr->segment.point2.y - (TICK_WIDTH / 2.0) * sin (angle);
           x2 = path_ptr->segment.point2.x + (TICK_WIDTH / 2.0) * cos (angle);
           y2 = path_ptr->segment.point2.y + (TICK_WIDTH / 2.0) * sin (angle);
           fprintf (file_ptr, " %f %f\n %f %f\n\" \"\n", x1, y1, x2, y2);
           if (TRACE) printf ("tickmark at end of current segment: \n");
           if (TRACE) printf (" %f %f\n %f %f\n\" \"\n", x1, y1, x2, y2);
        }
    }
    fclose (file_ptr);
    if (TRACE) printf ("\n*** graph_path complete\n");
    return;
}
```

```
/*------------------------------------------------------------------------*/

void   graph_world (circle_world, filename)

       /* Print circle_world data for unix 'graph' use, appended to filename */

       Circle_world  *circle_world;
       char          *filename;
{
    FILE        *file_ptr;
    int          i, j;                   /* indices                          */

    Circle_list *circle_ptr;         /* index pointer to current circle       */
    if (TRACE) printf ("\n*** graph_world start\n");
    circle_ptr = circle_world->circle_list; /* point to first circle in world */

    if ((file_ptr = fopen (filename, "a")) == ((FILE *) 0))
    {
        error ("graph_world:  file open failure!", NONFATAL);
        return;
    }

    if (TRACE) printf(" %f %f\n", circle_world->start.x, circle_world->start.y);
    if (TRACE) printf("\". Start\"\n");
    fprintf(file_ptr, " %f %f\n", circle_world->start.x, circle_world->start.y);
    fprintf(file_ptr, "\". Start\"\n");

    if (TRACE) printf (" %f %f\n", circle_world->goal.x, circle_world->goal.y);
    if (TRACE) printf ("\". Goal\"\n");
    fprintf(file_ptr, " %f %f\n", circle_world->goal.x, circle_world->goal.y);
    fprintf(file_ptr, "\". Goal\"\n");

    /* Loop to graph all circles in circle_world.                            */
    for (i=1; i <= circle_world->degree; ++i, circle_ptr = circle_ptr->next)
    {
        /* print current circle center                                       */
        if (TRACE) printf (" %f %f\n", circle_ptr->circle.center.x,
                                        circle_ptr->circle.center.y);
        fprintf (file_ptr, " %f %f\n", circle_ptr->circle.center.x,
                                        circle_ptr->circle.center.y);
        if (TRACE) printf ("\". Circle %d\"\n", i); /* label center w/ circle */
        fprintf (file_ptr, "\". Circle %d\"\n", i); /* label center w/ circle */
        /* print circle circumference at intervals   = 360 / SUBDIVISIONS     */
        for (j=0; j < 360 + (360 / SUBDIVISIONS); j += 360 / SUBDIVISIONS)
        {
            if (TRACE) printf (" %f %f\n",
                      (circle_ptr->circle.center.x  +
                         circle_ptr->circle.radius * cos (j * PI / 180.0)),
                      (circle_ptr->circle.center.y  +
                         circle_ptr->circle.radius * sin (j * PI / 180.0)));
            fprintf (file_ptr, " %f %f\n",
                      (circle_ptr->circle.center.x  +
                         circle_ptr->circle.radius * cos (j * PI / 180.0)),
                      (circle_ptr->circle.center.y  +
                         circle_ptr->circle.radius * sin (j * PI / 180.0)));
            if (circle_ptr->circle.radius == 0.0)
               break;                /* only one point needed in point case   */
        }
        if (TRACE) printf ("\" \"\n"); /* quoted blank to delimit this circle */
        fprintf (file_ptr, "\" \"\n"); /* quoted blank to delimit this circle */
    }
    fclose (file_ptr);
    if (TRACE) printf ("\n*** graph_world complete\n");
    return;
}
/*------------------------------------------------------------------------------*/

void   output_path (path, filename)

       /* Output path in AUV data file format, appended to filename           */

       Path  *path;
       char  *filename;
{
    FILE        *file_ptr,*k_ptr;
    int          i, j;               /* indices                              */
    Path_list   *path_ptr;           /* index pointer to legs on the path    */
    if (TRACE) printf ("\n*** output_path start\n");

    path_ptr = path->path_list;        /* point to first leg of path          */

    if ((file_ptr = fopen (filename, "a")) == ((FILE *) 0))
    {
        error ("output_path:  file open failure!", NONFATAL);
        return;
    }
    if ((k_ptr = fopen (K_FILENAME, "a")) == ((FILE *) 0))
    {
        error ("output_path:  file open failure!", NONFATAL);
        return;
    }

    if (path->label != NULL)
    {
        fprintf (file_ptr, "\nPath      %s\n\n", path->label); /* path header */
        if (TRACE) printf ("\nPath      %s\n\n", path->label); /* path header */
    }
```

```
else
{
    fprintf (file_ptr, "\nPath      \n\n");
    if (TRACE) printf ("\nPath      \n\n");
}

/* print starting line segment data                                        */
fprintf (file_ptr, "Segment  %8.2f %8.2f %8.2f %8.2f %8.2f %8.2f",
                    path->initial_segment.point1.x,
                    path->initial_segment.point1.y,
                    DEFAULT_Z,
                    path->initial_segment.point2.x,
                    path->initial_segment.point2.y,
                    DEFAULT_Z);
fprintf (k_ptr, " %8.2f %8.2f %8.2f %8.2f\n",
                    path->initial_segment.point1.x,
                    path->initial_segment.point1.y,
                    path->initial_segment.point2.x,
                    path->initial_segment.point2.y);
if (TRACE) printf ("Segment  %8.2f %8.2f %8.2f %8.2f %8.2f %8.2f",
                    path->initial_segment.point1.x,
                    path->initial_segment.point1.y,
                    DEFAULT_Z,
                    path->initial_segment.point2.x,
                    path->initial_segment.point2.y,
                    DEFAULT_Z);
pooltime++;
fprintf (file_ptr, "                time %4.1f\n", pooltime);
if (TRACE) printf ("                time %4.1f\n", pooltime);

/* print all succeeding arc / line segment combinations                    */
for (i=1; i <= path->degree; ++i, path_ptr = path_ptr->next)
{
 if (path_ptr->arc.rotation != 0)    /* don't print arc if nothing's there */
   {
    fprintf (file_ptr, "Arc      %8.2f %8.2f %8.2f %8.2f %8.2f %8.2f %2i ",
             path_ptr->arc.circle.center.x,
             path_ptr->arc.circle.center.y,
             DEFAULT_Z,
             path_ptr->arc.circle.radius,
             degrees (normalize2_rad (path_ptr->arc.angle1)),
             degrees (normalize2_rad (path_ptr->arc.angle2)),
             path_ptr->arc.rotation);

    fprintf (k_ptr, " %8.2f %8.2f %8.2f %2i\n ",
             path_ptr->arc.circle.radius,
             degrees (normalize2_rad (path_ptr->arc.angle1)),
             degrees (normalize2_rad (path_ptr->arc.angle2)),
             path_ptr->arc.rotation);
    if       (path_ptr->arc.rotation == CW)
              fprintf (file_ptr, "= CW      time %4.1f\n", pooltime);
    else if (path_ptr->arc.rotation == CCW)
              fprintf (file_ptr, "= CCW     time %4.1f\n", pooltime);
    else if (path_ptr->arc.rotation == CENTER)
              fprintf (file_ptr, "= CENTER  time %4.1f\n", pooltime);
    else      fprintf (file_ptr, "          time %4.1f\n", pooltime);

    if (TRACE) printf ("Arc      %8.2f %8.2f %8.2f %8.2f %8.2f %8.2f %2i ",
             path_ptr->arc.circle.center.x,
             path_ptr->arc.circle.center.y,
             DEFAULT_Z,
             path_ptr->arc.circle.radius,
             degrees (normalize2_rad (path_ptr->arc.angle1)),
             degrees (normalize2_rad (path_ptr->arc.angle2)),
             path_ptr->arc.rotation);

    if (TRACE)
    {
        if       (path_ptr->arc.rotation == CW)
                  printf ("= CW      time %4.1f\n", pooltime);
        else if (path_ptr->arc.rotation == CCW)
                  printf ("= CCW     time %4.1f\n", pooltime);
        else if (path_ptr->arc.rotation == CENTER)
                  printf ("= CENTER  time %4.1f\n", pooltime);
        else      printf ("          time %4.1f\n", pooltime);
    }
   }

    fprintf (file_ptr, "Segment  %8.2f %8.2f %8.2f %8.2f %8.2f %8.2f",
             path_ptr->segment.point1.x,
             path_ptr->segment.point1.y,
             DEFAULT_Z,
             path_ptr->segment.point2.x,
             path_ptr->segment.point2.y,
             DEFAULT_Z);
    fprintf (k_ptr, " %8.2f %8.2f %8.2f %8.2f\n",
             path_ptr->segment.point1.x,
             path_ptr->segment.point1.y,
             path_ptr->segment.point2.x,
             path_ptr->segment.point2.y);

    if (TRACE) printf ("Segment  %8.2f %8.2f %8.2f %8.2f %8.2f %8.2f",
                        path_ptr->segment.point1.x,
                        path_ptr->segment.point1.y,
                        DEFAULT_Z,
                        path_ptr->segment.point2.x,
                        path_ptr->segment.point2.y,
                        DEFAULT_Z);
    pooltime++;
    fprintf (file_ptr, "               time %4.1f\n", pooltime);
    if (TRACE) printf ("               time %4.1f\n", pooltime);

}
```

```
    fprintf (file_ptr, "\n");
    fprintf (k_ptr, "\n");
    if (TRACE) printf ("\n*** output_path complete\n");
    fclose  (file_ptr);
    fclose  (k_ptr);
    return;
}
/*-----------------------------------------------------------------------------*/

void   output_world (circle_world, filename)

       /* Output circle world using AUV data file format, appended to filename*/

       Circle_world *circle_world;
       char          *filename;
{
    FILE         *file_ptr,*k_ptr;
    int           i, j;               /* indices                              */

    Circle_list *circle_ptr;          /* index pointer to current circle      */
    circle_ptr = circle_world->circle_list; /* point to first circle in world */
    if (TRACE) printf ("\n*** output_world start\n");

    if ((file_ptr = fopen (filename, "a")) == ((FILE *) 0))
    {
        error ("output_world:  file open failure!", NONFATAL);
        return;
    }
    if ((k_ptr = fopen (K_FILENAME, "a")) == ((FILE *) 0))
    {
        error ("output_world:  file open failure!", NONFATAL);
        return;
    }
    fprintf (file_ptr,
                    "\n       Circle_World Shortest Path Determination\n\n");
    fprintf (file_ptr, "\nData specifications are according to ");
    fprintf (file_ptr, "the AUV Data Dictionary.\n\n\n");
    if (TRACE)
    {
       printf (        "\n       Circle_World Shortest Path Determination\n\n");
       printf (        "\nData specifications are according to ");
       printf (        "the AUV Data Dictionary.\n\n\n");
    }
    fprintf (file_ptr, "Point    %8.2f %8.2f %7.2f ", circle_world->start.x,
                                            circle_world->start.y,
                                            DEFAULT_Z);
    fprintf (k_ptr, " %8.2f %8.2f  %8.2f %8.2f  ", circle_world->start.x,
                                                  circle_world->start.y,
                                                  circle_world->goal.x,
                                                  circle_world->goal.y);

    fprintf (file_ptr, "    Start\n");
    if (TRACE) printf ("Point    %8.2f %8.2f %7.2f ", circle_world->start.x,
                                            circle_world->start.y,
                                            DEFAULT_Z);
    if (TRACE) printf ("    Start\n");

    fprintf (file_ptr, "Point    %8.2f %8.2f %7.2f ", circle_world->goal.x,
                                                   circle_world->goal.y,
                                                   DEFAULT_Z);
    fprintf (file_ptr, "    Goal\n\n");
    if (TRACE) printf ("Point    %8.2f %8.2f %7.2f ", circle_world->goal.x,
                                                   circle_world->goal.y,
                                                   DEFAULT_Z);
    if (TRACE) printf ("    Goal\n\n");

    if (TRACE) printf ("\n*** output_world circle_world->degree = %d \n",
                       circle_world->degree);
    for (i=1; i <= circle_world->degree; ++i, circle_ptr = circle_ptr->next)
    {
        /* print circle center and radius                                  */
        fprintf (file_ptr, "Circle  %8.2f %8.2f %8.2f %8.2f \n",
                            circle_ptr->circle.center.x,
                            circle_ptr->circle.center.y,
                            DEFAULT_Z,
                            circle_ptr->circle.radius);
        if (TRACE) printf ("Circle  %8.2f %8.2f %8.2f %8.2f \n",
                            circle_ptr->circle.center.x,
                            circle_ptr->circle.center.y,
                            DEFAULT_Z,
                            circle_ptr->circle.radius);
    }
    if (TRACE) printf ("\n*** output_world complete\n");
    fprintf (file_ptr, "\n");
    fprintf (k_ptr, "\n");
    fclose  (file_ptr);
    fclose  (k_ptr);
    return;
}
/*---------------------------------------------------------------------------*/

void center_graph_window (filename, xminptr, xmaxptr, yminptr, ymaxptr,
                          magnification)

/* Center (square off) the graph window so printed circles aren't distorted  */

     char   *filename;              /* graph filename for input & output     */
     double *xminptr, *xmaxptr,     /* output values (also appended to file) */
            *yminptr, *ymaxptr,
             magnification;         /* amount to magnify graph window bounds */
```

```
{
    FILE     *file_ptr;
    int       col;                       /* current column being used in line     */
    char      line [80];                 /* input line string of characters       */
    double    x, y;                      /* input values from current line        */
    double    xmin, xmax, ymin, ymax,    /* min/max values                        */
              deltax, deltay;            /* x, y max-min differences              */

    if (TRACE) printf ("\n*** center_graph_window start\n");

    xmin =   HUGE_VAL;                   /* Typical graph ticks are 5 units apart */
    ymin =   HUGE_VAL;
    xmax = - HUGE_VAL;
    ymax = - HUGE_VAL;

    if ((file_ptr = fopen (filename, "r")) == ((FILE *) 0))
    {
        error ("center_graph_window:  file initial open failure!", NONFATAL);
        return;
    }
    if (TRACE) printf ("\n*** center_graph_window:  %s is open\n", filename);

    while ((fgets (line, 81, file_ptr) != NULL))     /* read next line of file */
    {
           col = 0;
           while (line [col] == ' ') col++;          /* skip initial blanks     */
           if (isdigit(line[col]) || (line [col] == '-') || (line [col] == '+')
                                  || (line [col] == '.'))
           {
              if (sscanf (line+col,"%lf",&x) != 1) break; /* get x gracefully*/

              while (isdigit ((int) line [col])    || (line [col] == '-') ||
                             (line [col] == '.') || (line [col] == '+'))
                     col++;                        /* skip digits of x         */
              while ((line[col] == ' ') || (line [col] == ','))
                     col++;                        /* skip characters before y */

              if (sscanf (line+col,"%lf",&y) != 1) break;  /* get y gracefully*/
              if (xmin > x) xmin = x;
              if (ymin > y) ymin = y;
              if (xmax < x) xmax = x;
              if (ymax < y) ymax = y;
              if (TRACE)printf("\n*** center_graph_window loop check:");
              if (TRACE)printf("  (x, y)=(%6.2f, %6.2f)   ", x, y);
              if (TRACE)printf("  (xmin, ymin)=(%6.2f, %6.2f)", xmin, ymin);
              if (TRACE)printf("  (xmax, ymax)=(%6.2f, %6.2f)", xmax, ymax);
           } /* only lines beginning with numeric values are checked           */
    } /* end while */

    if (TRACE) printf ("\n*** center_graph_window while loop done\n          ");
    if (TRACE) printf ("(xmin, ymin) = (%6.2f, %6.2f)   ", xmin, ymin);
    if (TRACE) printf ("(xmax, ymax) = (%6.2f, %6.2f)   ", xmax, ymax);

    /* Now square off the extremes so no distortion occurs                     */
    if       ((ymax - ymin) < (xmax - xmin))
              ymax = ymin  + (xmax - xmin);
    else if ((xmax - xmin) < (ymax - ymin))
              xmax = xmin  + (ymax - ymin);
    if (magnification != 1.0) /* stretch out graph window boundaries          */
    {
       deltax = xmax - xmin;
       deltay = ymax - ymin;
       xmin -= deltax * (magnification - 1.0) / 2.0;
       xmax += deltax * (magnification - 1.0) / 2.0;
       ymin -= deltay * (magnification - 1.0) / 2.0;
       ymax += deltay * (magnification - 1.0) / 2.0;
    }

    if (TRACE) printf ("\n*** center_graph_window square-off ");
    if (TRACE) printf ("and magnification complete:\n        ");
    if (TRACE) printf ("(xmin, ymin) = (%6.2f, %6.2f)   ", xmin, ymin);
    if (TRACE) printf ("(xmax, ymax) = (%6.2f, %6.2f)   ", xmax, ymax);
    if (TRACE) printf ("\n        ");
    if (TRACE) printf ("magnification = %4.2f", magnification);

    *xminptr = xmin;        /* set returned values using pointer indirection   */
    *yminptr = ymin;
    *xmaxptr = xmax;
    *ymaxptr = ymax;

    fclose (file_ptr);
    if ((file_ptr = fopen (filename, "a")) == ((FILE *) 0))
    {
        error ("center_graph_window:  file re-open failure!", NONFATAL);
        return;
    }
    /* append min/max points to file to square off graph boundaries            */
    if ((xmin !=   HUGE_VAL) && (ymin !=   HUGE_VAL))
        fprintf (file_ptr, " %8.2f %8.2f\n\" \"\n", xmin, ymin);
    if ((xmax != - HUGE_VAL) && (ymax != - HUGE_VAL))
        fprintf (file_ptr, " %8.2f %8.2f\n\" \"\n", xmax, ymax);

    fclose (file_ptr);
    if (TRACE) printf ("\n*** center_graph_window complete\n");

    return;
}
/*----------------------------------------------------------------------------*/

void retrieve_circle_world (infilename,circle_world)
    char          infilename [40];
    Circle_world *circle_world;
{
    char          line [120];
    FILE         *file_ptr;
    double        x, y, r;
    Point         start_point, goal_point, center_point;
    Circle        circle;
    if (TRACE) printf ("\n*** retrieve_circle_world begin\n");

/*    printf ("\n\nEnter the name of the circle_world file to retrieve:  ");
    scanf ("%s", infilename);  remove keyboard input (bjl) */

    while ((file_ptr = fopen (infilename, "r")) == ((FILE *) 0))
    {
        error ("retrieve_circle_world file open failure...\n  ", NONFATAL);
       if (TRACE) printf ("\nPlease reenter the name of the circle_world file ");
       if (TRACE) printf ("to be retrieved:  ");
        scanf ("%s", infilename);
    }

    while (TRUE) /* loop to get start point */
    {
       if   (fscanf (file_ptr, "%s", line) == EOF) /* read start point */
       {
           error ("retrieve_circle_world start point read failure", NONFATAL);
           return;
       }
       else  if (strcmp (line, "Point") == 0)
             {
                 fscanf (file_ptr, "%lf %lf", &x, &y);
                 start_point = make_point (x, y);
                 if (TRACE)
                     printf ("\n*** Start point = (%4.2f, %4.2f)\n", x, y);
                 break;
             }
    }
    while (TRUE) /* loop to get goal point */
    {
       if   (fscanf (file_ptr, "%s", line) == EOF) /* read goal point */
       {
           error ("retrieve_circle_world goal point read failure", NONFATAL);
           return;
       }
       else  if (strcmp (line, "Point") == 0)
             {
                 fscanf (file_ptr, "%lf %lf", &x, &y);
                 goal_point = make_point (x, y);
                 if (TRACE)
                     printf ("\n*** Goal point  = (%4.2f, %4.2f)\n", x, y);
                 break;
             }
    }
    create_circle_world (start_point, goal_point, circle_world);

    while (TRUE) /* loop to get next circle */
    {
       if   (fscanf (file_ptr, "%s", line) == EOF) /* read next circle */
       {
           break;
       }
```

```
  else  if (strcmp (line, "Circle") == 0)
              {
                  fscanf (file_ptr, "%lf %lf %*lf %lf", &x, &y, &r);
                  center_point = make_point (x, y);
/*    Add 4.0 to circle radius as a safety distance.                     */
                  circle       = make_circle (center_point, r + 4.0);
                  if (TRACE)
                  printf ("\n*** Circle     = (%4.2f, %4.2f, %4.2f)\n", x, y, r);
                  add_circle_to_world (circle, circle_world);
              }
    }
    if (TRACE)
    {
        printf ("\n*** circle_world start point = (%4.2f, %4.2f)\n",
                  circle_world->start.x, circle_world->start.y);
        printf ("\n*** circle_world goal  point = (%4.2f, %4.2f)\n",
                  circle_world->goal.x, circle_world->goal.y);
        printf ("\n*** circle_world degree      = %d\n",
                  circle_world->degree);
    }
    if (TRACE) printf ("\n*** retrieve_circle_world complete\n");
    fclose (file_ptr);

    return;
}
/*----------------------------------------------------------------------------*/
```

APPENDIX C. EXPERT SYSTEM SOURCE CODE

Code associated with the expert system is included below. The files included are specmission, mission.c, controller.script, moss_info.pl, mission.pl.

```
/*      Specmission Expert system code

        Authors: Duane Davis, Brad Leonhardt

        Date:   18 March 1996

        Running:
                        >prowindows
                        >[specmission].
                        >go.

*/

:- ensure_loaded(library(math)).
:- ensure_loaded(mehelp), ensure_loaded(vehicle_info).
:- unknown(A,fail).
:- dynamic phase/5, start_phase/1, phase_list/1, current_phase/1,
  complete_successor/1, entry_mode/1, pathobject/1, abort_successor/1,
  x_scale/1, x_zero/1, y_scale/1, y_zero/1.

go :- quit(_,_), abolish(phase/5), abolish(start_phase/1),
        abolish(phase_list/1),abolish(entry_mode/1),
            asserta(phase_list([])), initial_menu.

initial_menu :- make_main_menu, make_phase_menu,
  send(@phase_dialog,open).

/* make_main_menu creates a dialog box with choices for creating, modifying,
/* and deleting phases, and for getting means end help to plan a mission.
/* A chart of the operating area is also displayed with point and click
/* capability for entering navigation points              */
make_main_menu :- reset, \+object(@start_dialog),
  new(@start_dialog,dialog('Options')),
  new(Start_menu,menu('Available Operations:          ',
     marked,initial_choice)),
  new(Quit,button('Quit',quit)),
  new(Make,button('Generate Mission Code',spec_complete)),
  new(@file_name,text_item('Output File Name:  ','',none)),
  new(Maps,menu('Available Charts:  ',cycle,get_map)),
  send(Maps,append,['Moss Landing','Test Tank']),
  new(Label,label('Coordinates from upper left corner.')),
  new(P, path), assertz(pathobject(P)), send(P,pen,2),
  new(@path_x,text_item('X:  ',0,0)),
  new(@path_y,text_item('Y:  ',0,0)),
  new(@path_x1,text_item('Chart X:  ',0,0)),
    new(@path_y1,text_item('Chart Y:  ',0,0)),
    new(Path_clear,button('Clear Path',path_clear)),
    new(@picture,picture), send(@picture,size,size(400,800)),
    new(@bmp,bitmap(400,800)), send(@bmp,load,rowe),
    send(@picture,clear), send(@picture,display,@bmp),
    send(@start_dialog,size,size(800,150)),
    send(Start_menu,append,['Specify Phase','Modify Phase','Delete Phase',
        'Means End Help']),
    send(@start_dialog,append,Start_menu),
    send(@file_name,right,Start_menu), send(Maps,right,@file_name),
    send(Quit,below,Start_menu), send(Make,right,Quit),
    send(Path_clear,right,Make),
    send(@path_x1,right,Path_clear), send(@path_y1,right,@path_x1),
    send(@picture,below,@start_dialog), send(@start_dialog,open),
    send(@picture, cursor_x, message(@path_x,selection,0)),
    send(@picture, cursor_y, message(@path_y,selection,0)),
    send(@picture, left_up, cascade(@picture,path_first,0)).

/* make_phase_menu creates a dialog box which is used for displaying
/* the names of phases as they are specified.  Each phase is displayed
/* with its complete and abort successor phase names          */
make_phase_menu :- new(@phase_dialog,dialog('Phase Summary')),
   pad_to_30('Specified Phases',Phase_label),
   pad_to_30('Complete Successor',Comp_label),
   pad_to_30('Abort Successor',Abort_label),
   new(@phases,label(Phase_label)),
   new(@c_succ,label(Comp_label)),
   new(@a_succ,label(Abort_label)),
   send(@phase_dialog,append,@phases),send(@c_succ,right,@phases),
   send(@a_succ,right,@c_succ).
/* Alternates between moss landing and test tank maps depending on
/* Menu selection from main dialog box      */
get_map(_,'Moss Landing') :- send(@bmp,load,xymoss),
   send(@picture,display,@bmp), consult(moss_info).
get_map(_,'Test Tank') :- send(@bmp,load,tank),
   send(@picture,display,@bmp), consult(tank_info).


/* Menus for Creation, Deletion, and Modification of Phases */

/* Create a new phase */
/* create_phase makes a dialog box to get the type of phase
/* that the user wishes to enter.  Types of phases are:
/*  Change Depth, Transit, Hover, GPS Fix, Rotate AUV Search,
/*  and Rotate Sonar Search              */
create_phase :- reset, retract(entry_mode(X)), fail.
create_phase :- asserta(entry_mode(create)), fail.
create_phase :- new(@type_dialog,dialog('Phase Type')),
   new(Menu,menu('Press button for next phase type:',marked,phase_info)),
   new(PReset,button('Reset Phase',phase_reset)),
   new(TReset,button('Cancel',partial_reset)),
   new(Quit,button('Quit',quit)),
  send(Menu,append,['Depth Change','Transit','Hover','GPS Fix',
     'Rotate Sonar Search', 'Rotate AUV Search']),
  send(@type_dialog,append,Menu),
  send(PReset,below,Menu), send(TReset,right,PReset),
  send(Quit,right,TReset), send(@type_dialog,open).
/* Get input specific to different phase types */
/* phase_info is used when creating or modifying phases to get
/* information specific to each type of phase (as well as
/* information general to all types)          */
phase_info(_,_) :- reset, make_common_items, fail.
phase_info(_,'Depth Change') :- asserta(current_phase('Depth Change')),
  new(@parameter_dialog,dialog('Depth Change Parameters')),
  new(@new_depth,text_item('New Depth:  ','',none)),
  send(@parameter_dialog,append,@phase_name),
  send(@new_depth,below,@phase_name),
  send(@time_out,below,@new_depth), display_common_items.
phase_info(_,'Transit') :- asserta(current_phase('Transit')),
  new(@parameter_dialog,dialog('Transit Parameters')),
  send(@parameter_dialog,append,@phase_name),
  make_x_y_depth_items,
  display_x_y_depth_items,
  send(@time_out,below,@new_depth), display_common_items.
phase_info(_,'Hover') :- asserta(current_phase('Hover')),
  new(@parameter_dialog,dialog('Hover Parameters')),
  send(@parameter_dialog,append,@phase_name),
  new(@heading,text_item('Heading:  ','',none)),
  make_x_y_depth_items,
  display_x_y_depth_items,
  send(@heading,below,@new_y),
  send(@time_out,below,@new_depth), display_common_items.
phase_info(_,'GPS Fix') :- asserta(current_phase('GPS Fix')),
  new(@parameter_dialog,dialog('GPS Fix Parameters')),
  send(@parameter_dialog,append,@phase_name),
  send(@time_out,below,@phase_name), display_common_items.
phase_info(_,'Rotate Sonar Search') :-
  asserta(current_phase('Rotate Sonar Search')),
  new(@parameter_dialog,dialog('Rotate Sonar Search Parameters')),
  send(@parameter_dialog,append,@phase_name),
  make_x_y_depth_items, display_x_y_depth_items,
  send(@time_out,below,@new_y), display_common_items.
phase_info(_,'Rotate AUV Search') :- asserta(current_phase('Rotate AUV Search')),
  new(@parameter_dialog,dialog('Rotate AUV Search Parameters')),
  send(@parameter_dialog,append,@phase_name),
  make_x_y_depth_items, display_x_y_depth_items,
  send(@time_out,below,@new_y), display_common_items.
phase_info(_,_) :- display_common_items.

/* make_common_items, display_common_items, make_x_y_depth_items, and
/* display_x_y_depth_items are used for creating and displaying buttons
/* and text items for information and operations common to multiple
/* types of phases            */
make_common_items :- new(@phase_name,text_item('Phase Name:  ','',none)),
  new(@time_out,text_item('Time Out:  ',500,none)),
  new(@done,button('Done',assert_phase)),
  new(@reset,button('Reset Phase',phase_reset)),
  new(@quit,button('Quit',quit)).

display_common_items :- phase_abort_successor(Abort_successor),
  phase_complete_successor(Complete_successor),
  send(Complete_successor,below,@time_out),
  send(Abort_successor,right,Complete_successor),
  send(@done,below,Complete_successor), send(@reset,right,@done),
  send(@quit,right,@reset), send(@parameter_dialog,open).



make_x_y_depth_items :- new(@new_depth,text_item('Depth:  ','',none)),
  get(@path_x1,selection,X), get(@path_y1,selection,Y),
  new(@new_x,text_item('X Position:  ',X,none)),
  new(@new_y,text_item('Y Position:  ',Y,none)).

display_x_y_depth_items :- send(@new_depth,below,@phase_name),
  send(@new_x,below,@new_depth),
  send(@new_y,below,@new_x).


/* phase_abort_successor and phase_complete_successor create menus for entering
/* the phases that will follow the phase being specified or modified */
phase_abort_successor(Abort_successor) :-
new(Abort_successor,menu('Phase Abort Successor',marked,handle_abort_successor)),
  phase_list(List),append(['Unspecified' | List],
                  ['mission_complete','mission_abort'],List2),
  send(Abort_successor,append,List2).

phase_complete_successor(Complete_successor) :-
  new(Complete_successor,menu('Phase Complete Successor',marked,
                      handle_complete_successor)),
  phase_list(List),append(['Unspecified' | List],
                  ['mission_complete','mission_abort'],List2),
  send(Complete_successor,append,List2).

handle_abort_successor(_,_) :- retract(abort_successor(X)), fail.
handle_abort_successor(_,'Unspecified') :- get_unspecified_phase(abort).
handle_abort_successor(_,Successor) :- asserta(abort_successor(Successor)).

handle_complete_successor(_,_) :- retract(complete_successor(X)), fail.
handle_complete_successor(_,'Unspecified') :- get_unspecified_phase(complete).
handle_complete_successor(_,Successor) :- asserta(complete_successor(Successor)).

/* Modify a previously specified phase */
/* modify_phase provides a menu with the names of all specified phases
/* for the user to choose from           */
modify_phase :- reset, retract(entry_mode(X)), fail.
modify_phase :- asserta(entry_mode(modify)), fail.
modify_phase :- new(@modify_dialog,dialog('Phase Modification')),
  new(Menu,menu('Press button for phase to modify',marked,modify_phase)),
  phase_list(Phases), send(Menu,append,Phases),
  new(Reset,button('Reset',partial_reset)),
  new(Quit,button('Quit',quit)),
  send(@modify_dialog,append,Menu),
  send(Reset,below,Menu), send(Quit,right,Reset),
  send(@modify_dialog,open).

modify_phase(_,Phase) :- phase(Phase,Type,Parameters,CSuccessor,ASuccessor),
  phase_info(_,Type), asserta(complete_successor(CSuccessor)),
  asserta(abort_successor(ASuccessor)),
      replace_parameters(Phase,Type,Parameters).



/* Replace parameters marshals all of the previously specified parameters
/* for a phase and places it in the appropriate places in the data entry
/* dialog when a phase is being modified so that the data does not have to
/* be entered from scratch          */
replace_parameters(Name,_,_) :- send(@phase_name,selection,Name), fail.
replace_parameters(_,'Depth Change',[Depth, Time_out]) :-
  send(@new_depth,selection,Depth), send(@time_out,selection,Time_out).
replace_parameters(_,'Transit',[X, Y, Depth, Time_out]) :-
  send(@new_x,selection,X), send(@new_y,selection,Y),
  send(@new_depth,selection,Depth), send(@time_out,selection,Time_out).
replace_parameters(_,'Hover',[X, Y, Depth, Heading, Time_Out]) :-
  send(@new_x,selection,X), send(@new_y,selection,Y), send(@new_depth,selection,Depth),
  send(@heading,selection,Heading), send(@time_out,selection,Time_out).
replace_parameters(_,'GPS Fix',[Time_out]) :-
  send(@time_out,selection,Time_out).
replace_parameters(_,'Rotate Sonar Search',[X, Y, Depth, Time_out]) :-
  send(@new_x,selection,X), send(@new_y,selection,Y),
  send(@new_depth,selection,Depth), send(@time_out,selection,Time_out).
replace_parameters(_,'Rotate AUV Search',[X, Y, Depth, Time_out]) :-
  send(@new_x,selection,X), send(@new_y,selection,Y),
  send(@new_depth,selection,Depth), send(@time_out,selection,Time_out).


/* Delete a previously specified phase */
/* delete_phase provides a list of all specified phases for the user
/* to choose from            */
delete_phase :- reset, retract(entry_mode(X)), fail.
delete_phase :- asserta(entry_mode(delete)), fail.
delete_phase :- new(@delete_dialog,dialog('Phase Deletion')),
  new(Menu,menu('Press button for phase to delete',marked,delete_phase)),
  phase_list(Phases), send(Menu,append,Phases),
  new(Reset,button('Reset',partial_reset)),
  new(Quit,button('Quit',quit)),
  send(@delete_dialog,append,Menu),
  send(Reset,below,Menu), send(Quit,right,Reset),
  send(@delete_dialog,open).
delete_phase(_,Phase) :- retract(phase(Phase,_,_,_,_)),
  retract(phase_list(PList)), delete(Phase,PList,NewPList),
  asserta(phase_list(NewPList)), reset.
% Append new point to most recent path when left button goes up;
% first path object will be the most recent
path_first(Picture, Pos) :-
  pathobject(P), !, send(P, append, Pos),
  get(@path_x,selection,X),string_to_num(X,Xnum),
  x_zero(X_zero),x_scale(X_scale),
  Xcorrect is (((Xnum - X_zero) / (X_scale)) + 0.5),
  get(@path_y,selection,Y),string_to_num(Y,Ynum),
  y_zero(Y_zero), y_scale(Y_scale),
  Ycorrect is (((Ynum - Y_zero) / (Y_scale)) + 0.5),
  floor(Xcorrect,Xrounded),
  floor(Ycorrect,Yrounded),
  send(@path_x1, selection, Yrounded),
  send(@path_y1, selection, Xrounded),
  to_menus(Xrounded,Yrounded),
  send(@picture, display, P).
%   send(@picture, left_up,message(P,append,0)).

to_menus(Xrounded,Yrounded) :- object(@new_x),
  send(@new_x, selection, Yrounded),
  send(@new_y, selection, Xrounded).
to_menus(Xrounded,Yrounded).

% clear the screen and eliminate all path objects
path_clear(_,_) :-
  send(@picture, clear), destroy_paths,
  send(@picture,display,@bmp),
  new(P, path),asserta(pathobject(P)),
  send(P, pen, 2),
  send(@picture, left_up, cascade(@picture,path_first,0)),
  send(@picture,display,@bmp).

destroy_paths :- retract(pathobject(P)), send(P,destroy), fail.
destroy_paths.

/* If a successor phase has not been specified its name is entered here */
/* The user is asked to enter the name of the unspecified phase.  In
/* order for code to be generated by the program, the named phase must
/* be specified later          */
get_unspecified_phase(abort) :- new(@ok1,button('OK',abort_ok)), fail.
get_unspecified_phase(complete) :- new(@ok1,button('OK',complete_ok)), fail.
get_unspecified_phase(_) :- new(@name_entry,dialog('Unspecified Phase Name')),
  new(Label1,label('Please enter the intended name of the unspecified phase')),
  new(Label2,
    label('You have to specify this phase before a mission generation')),
  new(@name,text_item('Name:  ','',none)),
  send(@name_entry,append,Label1), send(Label2,below,Label1),
  send(@name,below,Label2), send(@ok1,below,@name), send(@name_entry,open).
```

```
abort_ok(_,_) :- get(@name,selection,Name), asserta(abort_successor(Name)),
  send(@name_entry,destroy).
complete_ok(_,_) :- get(@name,selection,Name),
    asserta(complete_successor(Name)),
        send(@name_entry,destroy).


/* Callbacks for menus and pushbuttons */

initial_choice(_,'Means End Help') :- means_end_menu.
initial_choice(_,'Specify Phase') :- create_phase.
initial_choice(_,_) :- \+phase(_,_,_,_,_), invalid_option_report(no_phases).
initial_choice(_,'Modify Phase') :- modify_phase.
initial_choice(_,'Delete Phase') :- delete_phase.

/* Reset destroys all objects and facts associated with entry of a phase */
reset :- abolish(current_phase/1), abolish(abort_successor/1),
  abolish(complete_successor/1), fail.
reset :- object(@medialog), send(@medialog,destroy), fail.
reset :- object(@pointdialog), send(@pointdialog,destroy), fail.
reset :- object(@type_dialog), send(@type_dialog,destroy), fail.
reset :- object(@parameter_dialog), send(@parameter_dialog,destroy), fail.
reset :- object(@delete_dialog), send(@delete_dialog,destroy), fail.
reset :- object(@modify_dialog), send(@modify_dialog,destroy), fail.
reset.

phase_reset(_,_) :- reset, create_phase.
partial_reset(_,_) :- reset.

/* quit destroys all objects that are currently in existence */
quit(_,_) :- ok_error, fail.
quit(_,_) :- object(@path_x), send(@path_x,destroy), fail.
quit(_,_) :- object(@path_y), send(@path_y,destroy), fail.
quit(_,_) :- object(@mesolution), send(@mesolution,destroy), fail.
quit(_,_) :- object(@phase_dialog), send(@phase_dialog,destroy), fail.
quit(_,_) :- reset, object(@picture), send(@picture,destroy), fail.
quit(_,_) :- object(@bmp), send(@bmp,destroy), fail.
quit(_,_).

/* Callback when mission specification complete */
spec_complete(_,_) :- \+phase(_,_,_,_,_), invalid_option_report(no_phases).
spec_complete(_,_) :- reset, new(@first_phase,dialog('Start Phase')),
  new(Phase_menu,menu('Select Desired First Phase: ',marked,assert_start)),
  phase_list(Phase_list), send(Phase_menu,append,Phase_list),
  send(@first_phase,append,Phase_menu), send(@first_phase,open).

assert_start(_,_) :- retract(start_phase(X)), fail.
assert_start(_,Start) :- asserta(start_phase(Start)),
    send(@first_phase,destroy),parse_mission.
/* assert_phase asserts a fact of the form: */
/* phase(Name, Type, Parameter_list, Complete_successor, Abort_successor) */
/* for each phase specified by the user */
```

```
assert_phase(_,_) :- invalid_phase(Error), !, invalid_phase_report(Error).
assert_phase(_,_) :- get(@phase_name,selection,Phase_name),
%  delete_phase(_,Phase_name), fail.
  retract(phase(Phase_name,_,_,_,_)),
  retract(phase_list(PList)), delete(Phase_name,PList,NewPList),
  asserta(phase_list(NewPList)), fail.
assert_phase(_,_) :- current_phase(Phase),
  phase_parameters(Phase,Parameters), get(@phase_name,selection,Phase_name),
  complete_successor(CSuccessor),abort_successor(ASuccessor),
  asserta(phase(Phase_name,Phase,Parameters,CSuccessor,ASuccessor)),
  retract(phase_list(Phase_list)),
  asserta(phase_list([Phase_name | Phase_list])),
  pad_to_30(Phase_name,Name_string), pad_to_30(CSuccessor,CSucc_string),
  pad_to_30(ASuccessor,ASucc_string),
  new(Phase_label,label(Name_string)),new(CSucc_label,label(CSucc_string)),
  new(ASucc_label,label(ASucc_string)), send(Phase_label,below,@phases),
  send(CSucc_label,right,Phase_label), send(ASucc_label,right,CSucc_label),
  reset.
/* phase_parameters marshals parameters required for different phase types */
/* into a list for inclusion in the phase facts asserted by assert_phase */
phase_parameters('Depth Change',[Depth, Time_out]) :-
  get(@new_depth,selection,SDepth), string_to_num(SDepth,Depth),
  get(@time_out,selection,STime_out), string_to_num(STime_out,Time_out).
phase_parameters('Transit',[X, Y, Depth, Time_out]) :-
  get(@new_x,selection,SX), string_to_num(SX,X),
  get(@new_y,selection,SY), string_to_num(SY,Y),
  get(@new_depth,selection,SDepth), string_to_num(SDepth,Depth),
  get(@time_out,selection,STime_out), string_to_num(STime_out,Time_out).
phase_parameters('Hover',[X, Y, Depth, Heading, Time_out]) :-
  get(@new_x,selection,SX), string_to_num(SX,X),
  get(@new_y,selection,SY), string_to_num(SY,Y),
  get(@heading,selection,SHeading), string_to_num(SHeading,Heading),
  get(@new_depth,selection,SDepth), string_to_num(SDepth,Depth),
  get(@time_out,selection,STime_out), string_to_num(STime_out,Time_out).
phase_parameters('GPS Fix',[Time_out]) :-
  get(@time_out,selection,STime_out), string_to_num(STime_out,Time_out).
phase_parameters('Rotate Sonar Search',[X, Y, Depth, Time_out]) :-
  get(@new_x,selection,SX), string_to_num(SX,X),
  get(@new_y,selection,SY), string_to_num(SY,Y),
  get(@new_depth,selection,SDepth), string_to_num(SDepth,Depth),
  get(@time_out,selection,STime_out), string_to_num(STime_out,Time_out).
phase_parameters('Rotate AUV Search',[X, Y, Depth, Time_out]) :-
  get(@new_x,selection,SX), string_to_num(SX,X),
  get(@new_y,selection,SY), string_to_num(SY,Y),
  get(@new_depth,selection,SDepth), string_to_num(SDepth,Depth),
  get(@time_out,selection,STime_out), string_to_num(STime_out,Time_out).


/* Phase and Mission Rules and Parsers */

/* Rules for determining when a specified phase is invalid */
invalid_phase(no_name) :- get(@phase_name,selection,'').
invalid_phase(duplicate_phase) :- entry_mode(create),
  get(@phase_name,selection,Name), phase(Name,_,_,_,_).
invalid_phase(no_successor) :- \+abort_successor(X).
invalid_phase(no_successor) :- abort_successor('').
invalid_phase(no_successor) :- \+complete_successor(X).
invalid_phase(no_successor) :- complete_successor('').
invalid_phase(non_number) :- object(@time_out),get(@time_out,selection,String),
  \+string_to_num(String,_).
invalid_phase(non_number) :- object(@new_x),get(@new_x,selection,String),
  \+string_to_num(String,_).
invalid_phase(non_number) :- object(@new_y),get(@new_y,selection,String),
  \+string_to_num(String,_).
invalid_phase(non_number) :- object(@new_depth),
   get(@new_depth,selection,String),
  \+string_to_num(String,_).
invalid_phase(non_number) :- object(@heading),get(@heading,selection,String),
  \+string_to_num(String,_).
invalid_phase(too_deep) :- object(@new_depth), max_vehicle_depth(Deepest),
  get(@new_depth,selection,SDepth), string_to_num(SDepth,Depth),
  Depth > Deepest.
invalid_phase(bottom_hit) :- object(@new_depth), max_area_depth(Bottom),
  get(@new_depth,selection,SDepth), string_to_num(SDepth,Depth),
  Depth > Bottom.
invalid_phase(too_shallow) :- object(@new_depth),
  get(@new_depth,selection,SDepth), string_to_num(SDepth,Depth),
  Depth < 0.
invalid_phase(out_of_area) :- object(@new_x), get(@new_x,selection,SX),
  string_to_num(SX,X), op_area(X1,_,X2,_),
  (X < X1; X > X2).
invalid_phase(out_of_area) :- object(@new_y), get(@new_y,selection,SY),
  string_to_num(SY,Y), op_area(_,Y1,_,Y2),
  (Y < Y1; Y > Y2).
invalid_phase(bottom_hit) :- object(@new_depth),
  get(@new_depth,selection,SDepth),
  object(@new_x), get(@new_x,selection,SX), object(@new_y),
  get(@new_y,selection,SY), string_to_num(SDepth,Depth),
  string_to_num(SX,X), string_to_num(SY,Y),
  area_depth(X1,Y1,X2,Y2,Area_depth), Y >= Y1, Y =< Y2, X >= X1, X =< X2,
  Area_depth < Depth.

/* Check mission for errors and generate code if mission valid*/
parse_mission :- reset_phase_error, fail.
parse_mission :- mission_error(Phase,loop), !,
  invalid_mission_report(Phase,loop).
parse_mission :- phase(Phase,_,_,_,_),
  setof(Error,mission_error(Phase,Error),Error_list), !,
  ok_error(_,_), phase_error_report(Phase,Error_list), fail.
parse_mission :- tell(command_strings), fail.
parse_mission :- get(@file_name,selection,File_name),
  start_phase(Start_phase),
  list_concat(['file_name ',File_name,' ',Start_phase],Command),
  write(Command),nl, fail.
```

```
parse_mission :- phase(Phase,_,_,_,_), generate_code(Phase), fail.
parse_mission :- told, quit(_,_), fail.
parse_mission :- unix(shell('mission')).


/* Generate code writes a series of strings to a file called "mission_strings"
/* These strings are used by the C program to generate code in whatever
/* language is required.  The current version generates Prolog code.   */
generate_code(Phase) :- phase(Phase,'Depth Change',
  [Depth,Time_out],CSucc,ASucc),
  list_concat(['depth_change ',Phase,' ',CSucc,' ',ASucc,' ',
    Time_out,' ',Depth],Command),
  write(Command),nl.
generate_code(Phase) :- phase(Phase,'Transit',[X,Y,Depth,Time_out],CSucc,ASucc),
  list_concat(['waypoint ',Phase,' ',CSucc,' ',ASucc,' ',
    Time_out,' ',X,' ',Y,' ',Depth],Command),
  write(Command),nl.
generate_code(Phase) :- phase(Phase,'Hover',[X,Y,Depth,Heading,Time_out],
    CSucc,ASucc),
  list_concat(['hoverpoint ',Phase,' ',CSucc,' ',ASucc,' ',
    Time_out,' ',X,' ',Y,' ',Depth,' ',Heading],Command),
  write(Command),nl.
generate_code(Phase) :- phase(Phase,'GPS Fix',[Time_out],CSucc,ASucc),
  list_concat(['get_gps_fix ',Phase,' ',CSucc,' ',ASucc,' ',Time_out],Command),
  write(Command),nl.
generate_code(Phase) :-
   phase(Phase,'Rotate Sonar Search',[X,Y,Depth,Time_out],CSucc,ASucc),
  list_concat(['rotate_sonar_search ',Phase,' ',CSucc,' ',ASucc,' ',
    Time_out,' ',X,' ',Y,' ',Depth,' '],Command),
  write(Command),nl.
generate_code(Phase) :- phase(Phase,'Rotate AUV Search',
   [X,Y,Depth,Time_out],CSucc,ASucc),
  list_concat(['sonar_search ',Phase,' ',CSucc,' ',ASucc,' ',
    Time_out,' ',X,' ',Y,' ',Depth,' '],Command),
  write(Command),nl.


/* Rules for finding errors in a specified mission */
mission_error(Phase,loop) :- phase(Phase,_,_,_,_), reachable(Phase,Phase).
mission_error(_,no_file) :- get(@file_name,selection,'').
mission_error(_,no_start) :- \+start_phase(_).
mission_error(_,no_complete) :- \+phase(_,_,_,_,'mission_complete'),
  \+phase(_,_,_,'mission_complete',_).



mission_error(Phase,dangle) :- phase(Phase,_,_,Successor,_),
  \+phase(Successor,_,_,_,_),
  \+same(Successor,'mission_abort'), \+same(Successor,'mission_complete').
mission_error(Phase,dangle) :- phase(Phase,_,_,_,Successor),
  \+phase(Successor,_,_,_,_),
  \+same(Successor,'mission_abort'), \+same(Successor,'mission_complete').
```

```
mission_error(Phase,unreachable) :- start_phase(Start), phase(Phase,_,_,_,_),
  \+same(Start,Phase), \+reachable(Start,Phase).
mission_error(Phase,short_timer) :- phase(Phase,_,Parameters,_,_),
  last(Parameters,Time_out), max_transit_distance(Phase,Dist),
  max_speed(Speed),
  Minimum_time is Dist / Speed, Minimum_time > Time_out.
/* Error Reporting Routines for Phase Errors and Mission Errors */

/* Modify or Delete phase selected when no phases have been specivied */
invalid_option_report(_) :- object(@error_window1),
  send(@error_window1,destroy), fail.
invalid_option_report(Error) :- error_code(Error,Message),
  new(@error_window1,dialog('Invalid Option')),
  new(OK,button('OK',ok_error)),
  new(Label,label(Message)),
  send(@error_window1,append,Label),
  send(OK,below,Label), send(@error_window1,open).


/* There is an error in the phase specification */
invalid_phase_report(_) :- object(@error_window2), send(@error_window2,destroy), fail.
invalid_phase_report(Code) :- error_code(Code,Message),
  new(@error_window2,dialog('Invalid Phase')),
  new(OK,button('OK',ok_error)),
  new(Label1,label('PHASE ERROR:  THE SPECIFIED PHASE IS INVALID')),
  new(Label2,label(Message)),
  send(@error_window2,append,Label1), send(Label2,below,Label1),
  send(OK,below,Label2), send(@error_window2,open).


/* There is a loop in the mission specification, no further parsing can occur
/* until the loop is eliminated      */
invalid_mission_report(_,_) :- object(@error_window3),
  send(@error_window3,destroy), fail.
invalid_mission_report(Phase,Code) :- error_code(Code,Message),
  new(@error_window3,dialog('Mission Error')),
  new(OK,button('OK',ok_error)),
  new(Label1,label('MISSION ERROR IN PHASE: ')),
  new(Label2,label(Phase)),
  new(Label3,label(Message)),
  send(@error_window3,append,Label1), send(Label2,right,Label1),
  send(Label3,below,Label1), send(OK,below,Label3),
  send(@error_window3,open).




/* There is at least one error associated with a phase of the mission
/* All errors associated with a phase are displayed at once      */
phase_error_report(_,[]).
phase_error_report(Phase,_) :- \+object(@error_window4),
  new(@error_window4,dialog('Phase Errors')),
  new(@ok,button('OK',ok_error)),
  new(Label1,label('MISSION ERRORS IN PHASE: ')),
  new(Label2,label(Phase)), send(@error_window4,append,Label1),
  send(Label2,right,Label1), send(@ok,below,Label2),
  send(@error_window4,open), fail.
phase_error_report(Phase,[Error1 | Rest]) :- !, error_code(Error1,Message),
  new(Label,label(Message)),send(Label,above,@ok),
  phase_error_report(Phase,Rest).


/* Destroy error dialog box */
ok_error(_,_) :- object(@error_window1), send(@error_window1,destroy), fail.
ok_error(_,_) :- object(@error_window2), send(@error_window2,destroy), fail.
ok_error(_,_) :- object(@error_window3), send(@error_window3,destroy), fail.
ok_error(_,_) :- object(@error_window4), send(@error_window4,destroy).
ok_error(_,_).

reset_phase_error :- ok_error(_,_), object(@ok), send(@ok,destroy).
reset_phase_error.
/* Codes to match error names to an output message */
error_code(no_phases,'There are no specified phases in memory').
error_code(no_name,'You did not specify a name for the phase').
error_code(duplicate_phase,'A phase by that name has already been specified').
error_code(no_successor,'You did not specify one of the successor phases').
error_code(non_number,'You have entered a non number in a numerical field').
error_code(too_deep,'The depth you have entered is too deep for the vehicle').
error_code(bottom_hit,'The depth you specified is too deep for this area').
error_code(too_shallow,'Depth must be positive').
error_code(out_of_area,'Specified pointoutside the designated operating area').
error_code(unreachable,'Phase not reachable from start phase').
error_code(loop,'Phase reachable from itself').
error_code(dangle,'Successor Phase Undefined').
error_code(short_timer,'Specified time-out inadequate for phase completion').
error_code(no_complete,'No mission completion criteria specified').
error_code(no_start,'No mission starting phase specified').
error_code(no_file,'You did not specify an output file').

/* Utility Functions */
successor(Phase1,Phase2) :- phase(Phase1,_,_,Phase2,_).
successor(Phase1,Phase2) :- phase(Phase1,_,_,_,Phase2).

reachable(Phase1,Phase2) :- successor(Phase1,Phase2).
reachable(Phase1,Phase2) :- successor(Phase1,Phase3), reachable(Phase3,Phase2).

transit_distance(Phase,Distance) :- phase(Phase,Type1,[X1, Y1 | L1],_,_),
  \+same(Type1,'GPS Fix'), \+same(Type1,'Depth Change'),
  position_predecessor(Phase,Predecessor),
  phase(Predecessor,_,[X2, Y2 | L2],_,_),
  distance(X1,Y1,X2,Y2,Distance).
transit_distance(Phase,Distance) :- phase(Phase,Type1,[X1, Y1 | L],_,_),
  \+same(Type1,'GPS Fix'), \+same(Type1,'Depth Change'),
  start_phase(Start), start_position(X2,Y2,D), same(Start,Phase),
  distance(X1,Y1,X2,Y2,Distance).

max_transit_distance(Phase,Distance) :- bagof(Dist,transit_distance(Phase,Dist),
  DList),sort(DList,SortedList),last(SortedList,Distance).

position_predecessor(Phase,Predecessor) :-
  (phase(Predecessor,Type,_,Phase,_); phase(Predecessor,Type,_,_,Phase)),
  \+same(Type,'GPS Fix'), \+same(Type,'Depth Change').
position_predecessor(Phase,Predecessor) :-
  (phase(Phase2,Type,_,Phase,_); phase(Phase2,Type,_,_,Phase)),
  (same(Type,'GPS Fix'); same(Type,'Depth Change')),
  position_predecessor(Phase2,Predecessor).

distance(X1,Y1,X2,Y2,Distance):- X_plus_Y_Squared is
     (X2-X1)*(X2-X1) + (Y2-Y1)*(Y2-Y1),
  sqrt(X_plus_Y_Squared,Distance).

same(X,X).

string_to_num(String,Num) :- name(String,Asc),name(Num,Asc),number(Num).

last([X],X) :- !.
last([X | L],Y) :- last(L,Y).

concatenate(S1,S2,S) :- name(S1,AS1), name(S2,AS2), append(AS1,AS2,AS),
  name(S,AS).

list_concat([],'').
list_concat([X|L],S) :- list_concat(L,Part), concatenate(X,Part,S).

pad_to_30(String,String) :- name(String,List), length(List,Length),
  Length >= 30, !.
pad_to_30(String,New_string) :- concatenate(String,' ',Sub_string),
  pad_to_30(Sub_string,New_string).
```

```
/******************************************************************************/
/*
 Program:      mission.c    AUV strategic level program

 Authors:      Brad Leonhardt Duane Davis

 Revised:      21 January 96

 System:       SUN Voyager Solaris 2.4 OS; SGI Irix 5.3
 Compiler:      Sun C; IRIX cc

 Compilation:    cc mission.c -o mission

This code is used to create Prolog code to run the Phoenix Autonomous Vehicle
It can be run with the specmission prolog expert system which creates a data
file for use in the mission generator.

/******************************************************************************/


#include <string.h>
#include <stdio.h>
#include <ctype.h>

#define INPUT_FILE  "controller.script"       /*  Fixed Mission Controller    */
#define DATA_FILE   "command_strings"        /*  Expert System Output        */

/******************************************************************************/
/*                   function prototypes                                      */
/******************************************************************************/

int              main                                      ();
void             depth_change                   ();
void             hoverpoint                                ();
void             waypoint                                  ();
void             sonar_search                              ();
void             get_gps_fix                               ();
void             rotate_sonar_search                       ();
void         course                     ();
void             wait                                      ();
void             parse_command                             ();
void             time_out                                  ();
void             next_phase                                ();
void             fail_phase                                ();

FILE * file_ptr;
FILE * out_file_ptr;

char command      [200];
char out_file_name [200];
char shell_cmd    [200];
char phase_name   [200];
char nextphase    [200];
char failphase    [200];
char cmd          [200];

int variable1,variable2,variable3,variable4,variable5;
int i;


/****************************************************************************/
/*  Parse command breaks input string into variables to be used in program  */
/****************************************************************************/

void parse_command (cmd)
char * cmd;
{
   sscanf(cmd, "%s %s %s %s %d %d %d %d %d",
        command,phase_name,nextphase,failphase,
        &variable1,&variable2,&variable3,&variable4,&variable5);

/* printf ("COMMAND %s PHASE NAME %s VAR1 %d VAR2 %d VAR3 %d VAR4 %d VAR5 %d",
 command,phase_name,nextphase,failphase,
 variable1,variable2,variable3,variable4,variable5);
*/
}

/****************************************************************************/

int main ()

{
  char cmd [200];      /*       input line string of characters          */

  /*           Open file generated from expert system                    */

  if ((file_ptr = fopen (DATA_FILE, "r")) == ((FILE *) 0))
  {
    printf ("input_read_path:  file open failure!\n");
    return;
  }

  fgets (cmd,81,file_ptr);  /*       Read a line of data                 */

  parse_command (cmd);

  /*Create an output file for mission and copy mission controller data to it*/

  if (strcmp (command,"file_name") == 0)
  {
    strcpy(out_file_name,phase_name);
    strcat(shell_cmd,"cp controller.script ");
    strcat(shell_cmd,out_file_name);
    printf("Shell command :%s\n",shell_cmd);
  system(shell_cmd);
}

printf("Input file created\n");

if ((out_file_ptr = fopen (out_file_name, "a")) == ((FILE *) 0))
{
   printf("INVALID INPUT FILE NAME  %s\n",out_file_name);
   fclose  (file_ptr);
   return(0);
}

fprintf (out_file_ptr, "\t\t\t   asserta(current_phase(%s)),\n",nextphase);
fprintf (out_file_ptr, "\t\t\t   asserta(complete(0)),\n");
fprintf (out_file_ptr, "\t\t\t   asserta(abort(0)).\n\n");

while ((fgets (cmd, 81, file_ptr) != NULL))   /*    read next line      */
{
   printf("Input file string %s\n",cmd);
   parse_command(cmd);

  fprintf (out_file_ptr, "%%      %s\n\n\n",phase_name);
  fprintf (out_file_ptr, "execute_phase(%s)\n",phase_name);
  fprintf (out_file_ptr, "\t\t    :-   nl,printsc('PHASE");
  fprintf (out_file_ptr, " %s STARTED.'),\n",phase_name);

  /*          Determine which template to use                        */

  if (strcmp (command,"depth_change") == 0)
          depth_change ();

  else if (strcmp (command,"hoverpoint") == 0)
          hoverpoint ();

  else if (strcmp (command,"waypoint") == 0)
          waypoint ();

  else if (strcmp (command,"sonar_search") == 0)
          sonar_search ();

  else if (strcmp (command,"get_gps_fix") == 0)
          get_gps_fix ();

  else if (strcmp (command,"rotate_sonar_search") == 0)
          rotate_sonar_search ();

  else if (strcmp (command,"course") == 0)
     course ();

  else if (strcmp (command,"wait") == 0)
     wait ();
```

```
      else
      {
         printf("INVALID PHASE INPUT %s\n",command);
         fprintf (file_ptr, "\n");
         fclose  (file_ptr);
         fprintf (out_file_ptr, "\n");
         fclose  (out_file_ptr);
         return(0);
      }

   }


   fprintf (file_ptr, "\n");
   fclose  (file_ptr);
   fprintf (out_file_ptr, "\n");
   fclose  (out_file_ptr);
}

/******************************************************************************/
/*  Functions that produce the Prolog code for the various types of phases   */
/******************************************************************************/

void depth_change ()

{

   fprintf (out_file_ptr, "\t\t\tood('depth ");
   fprintf (out_file_ptr, "%d',X),X==1,\n",variable2);
   fprintf (out_file_ptr, "\t\t\t   printsc('depth %d'),\n",variable2);
   fprintf (out_file_ptr, "\t\t\tood('start_timer ");
   fprintf (out_file_ptr, "%d',X),X==1,\n",variable1);
   fprintf (out_file_ptr, "\t\t\t   repeat,");
   fprintf (out_file_ptr, "phase_completed(%s).\n\n",phase_name);

   fprintf (out_file_ptr, "phase_completed(%s)\n",phase_name);
   fprintf (out_file_ptr, "\t\t     :- ood('ask_depth_reached',X),X==1,\n");
   fprintf (out_file_ptr, "\t\t\t   printsc('DEPTH REACHED.'),\n");
   fprintf (out_file_ptr, "\t\t\t   asserta(complete(");
   fprintf (out_file_ptr, "%s)).\n\n",phase_name);

   time_out ();

   next_phase ();

   fail_phase ();

}

void hoverpoint ()

{
    fprintf (out_file_ptr, "\t\t\tood('hover ");
    fprintf (out_file_ptr, "%d %d %d %d',X),X==1,\n",
      variable2,variable3,variable4,variable5);
    fprintf (out_file_ptr, "\t\t\t   printsc('hover %d %d %d %d'),\n",
      variable2,variable3,variable4,variable5);
    fprintf (out_file_ptr, "\t\t\tood('start_timer ");
    fprintf (out_file_ptr, "%d',X),X==1,\n",variable1);
    fprintf (out_file_ptr, "\t\t\t   repeat,");
    fprintf (out_file_ptr, "phase_completed(%s).\n\n",phase_name);


    fprintf (out_file_ptr, "phase_completed(%s)\n",phase_name);
    fprintf (out_file_ptr, "\t\t     :- ood('ask_hoverpt_reached',X),X==1,\n");
    fprintf (out_file_ptr, "\t\t\t   printsc('HOVERPOINT REACHED.'),\n");
    fprintf (out_file_ptr, "\t\t\t   asserta(complete(");
    fprintf (out_file_ptr, "%s)).\n\n",phase_name);

    time_out ();

    next_phase ();

    fail_phase ();

}

void waypoint ()

{

    fprintf (out_file_ptr, "\t\t\tood('waypoint ");
    fprintf (out_file_ptr, "%d %d %d',X),X==1,\n",
      variable2,variable3,variable4);
    fprintf (out_file_ptr, "\t\t\t   printsc('waypoint %d %d %d'),\n",
      variable2,variable3,variable4);
    fprintf (out_file_ptr, "\t\t\tood('start_timer ");
    fprintf (out_file_ptr, "%d',X),X==1,\n",variable1);
    fprintf (out_file_ptr, "\t\t\t   repeat,");
    fprintf (out_file_ptr, "phase_completed(%s).\n\n",phase_name);


    fprintf (out_file_ptr, "phase_completed(%s)\n",phase_name);
    fprintf (out_file_ptr, "\t\t     :- ood('ask_waypt_reached',X),X==1,\n");
    fprintf (out_file_ptr, "\t\t\t   printsc('WAYPOINT REACHED.'),\n");
    fprintf (out_file_ptr, "\t\t\t   asserta(complete(");
    fprintf (out_file_ptr, "%s)).\n\n",phase_name);

    time_out ();

    next_phase ();

    fail_phase ();
```

```
}


void sonar_search ()

{

   fprintf (out_file_ptr, "\t\t\tood('sonar_search',X),X==1,\n");
   fprintf (out_file_ptr, "\t\t\t   printsc('sonar_search !'),\n");
   fprintf (out_file_ptr, "\t\t\tood('start_timer ");
   fprintf (out_file_ptr, "%d',X),X==1,\n",variable1);
   fprintf (out_file_ptr, "\t\t\t   repeat,");
   fprintf (out_file_ptr, "phase_completed(%s).\n\n",phase_name);


   fprintf (out_file_ptr, "phase_completed(%s)\n",phase_name);
   fprintf (out_file_ptr, "\t\t     :- ood('ask_sonar_search_complete',X),");
   fprintf (out_file_ptr, "X==1,\n");
   fprintf (out_file_ptr, "\t\t\t   printsc('SONAR SEARCH COMPLETE.'),\n");
   fprintf (out_file_ptr, "\t\t\t   asserta(complete(");
   fprintf (out_file_ptr, "%s)).\n\n",phase_name);

   time_out ();

   next_phase ();

   fail_phase ();

}


void get_gps_fix ()

{

   fprintf (out_file_ptr, "\t\t\tood('get_gps_fix',X),X==1,\n");
   fprintf (out_file_ptr, "\t\t\t   printsc('get_gps_fix !'),\n");
   fprintf (out_file_ptr, "\t\t\tood('start_timer ");
   fprintf (out_file_ptr, "%d',X),X==1,\n",variable1);
   fprintf (out_file_ptr, "\t\t\t   repeat,");
   fprintf (out_file_ptr, "phase_completed(%s).\n\n",phase_name);

   fprintf (out_file_ptr, "phase_completed(%s)\n",phase_name);
   fprintf (out_file_ptr, "\t\t     :- ood('ask_get_gps_fix',X),");
   fprintf (out_file_ptr, "X==1,\n");
   fprintf (out_file_ptr, "\t\t\t   printsc('G P S FIX OBTAINED.'),\n");
   fprintf (out_file_ptr, "\t\t\t   asserta(complete(");
   fprintf (out_file_ptr, "%s)).\n\n",phase_name);

   time_out ();

   next_phase ();
   fail_phase ();

}



void rotate_sonar_search ()

{

   fprintf (out_file_ptr, "\t\t\tood('rotate_search',X),X==1,\n");
   fprintf (out_file_ptr, "\t\t\t   printsc('rotate_search !'),\n");
   fprintf (out_file_ptr, "\t\t\tood('start_timer ");
   fprintf (out_file_ptr, "%d',X),X==1,\n",variable1);
   fprintf (out_file_ptr, "\t\t\t   repeat,");
   fprintf (out_file_ptr, "phase_completed(%s).\n\n",phase_name);


   fprintf (out_file_ptr, "phase_completed(%s)\n",phase_name);
   fprintf (out_file_ptr, "\t\t     :- ood('");
   fprintf (out_file_ptr, "ask_rotate_search_complete',X),X==1,\n");
   fprintf (out_file_ptr, "\t\t\t   printsc('ROTATE SEARCH COMPLETE.'),\n");
   fprintf (out_file_ptr, "\t\t\t   asserta(complete(");
   fprintf (out_file_ptr, "%s)).\n\n",phase_name);

   time_out ();

   next_phase ();

   fail_phase ();

}



void course ()
{
   fprintf (out_file_ptr, "\t\t\tood('course',X),X==1,\n");
   fprintf (out_file_ptr, "\t\t\t   printsc('course !'),\n");
   fprintf (out_file_ptr, "\t\t\tood('start_timer ");
   fprintf (out_file_ptr, "%d',X),X==1,\n",variable1);
   fprintf (out_file_ptr, "\t\t\t   repeat,");
   fprintf (out_file_ptr, "phase_completed(%s).\n\n",phase_name);


   fprintf (out_file_ptr, "phase_completed(%s)\n",phase_name);
   fprintf (out_file_ptr, "\t\t     :- ood('");
   fprintf (out_file_ptr, "ask_course_reached',X),X==1,\n");
   fprintf (out_file_ptr, "\t\t\t   printsc('COURSE CHANGE COMPLETE.'),\n");
   fprintf (out_file_ptr, "\t\t\t   asserta(complete(");
   fprintf (out_file_ptr, "%s)).\n\n",phase_name);
```

```
    time_out ();

    next_phase ();

    fail_phase ();

}


void wait ()


{
    fprintf (out_file_ptr, "\t\t\tood('start_timer ");
    fprintf (out_file_ptr, "%d',X),X==1,\n",variable1);
    fprintf (out_file_ptr, "\t\t\t   repeat,");
    fprintf (out_file_ptr, "phase_completed(%s).\n\n",phase_name);


    fprintf (out_file_ptr, "phase_completed(%s)\n",phase_name);
    fprintf (out_file_ptr, "\t\t     :- ood('");
    fprintf (out_file_ptr, "ask_time_out',X),X==1,\n");
    fprintf (out_file_ptr, "\t\t\t   printsc('WAIT COMPLETE.'),\n");
    fprintf (out_file_ptr, "\t\t\t   asserta(complete(");
    fprintf (out_file_ptr, "%s)).\n\n",phase_name);

    next_phase ();

}


void time_out ()
{

    fprintf (out_file_ptr, "phase_completed(%s)\n",phase_name);
    fprintf (out_file_ptr, "\t\t     :- ood('ask_time_out',X),X==1,\n");
    fprintf (out_file_ptr, "\t\t\t   printsc('PHASE %s ",phase_name);
    fprintf (out_file_ptr, "ABORTED DUE TO TIME OUT.'),\n");
    fprintf (out_file_ptr, "\t\t\t   asserta(abort(%s)).\n\n",phase_name);

}


void next_phase ()
{

    fprintf (out_file_ptr, "next_phase(%s)\n",phase_name);
    fprintf (out_file_ptr, "\t\t     :-    complete(%s),\n",phase_name);
    fprintf (out_file_ptr, "\t\t\t   retract(current_phase(");
    fprintf (out_file_ptr, "%s)),\n",phase_name);
    fprintf (out_file_ptr, "\t\t\t   asserta(current_phase(");
    fprintf (out_file_ptr, "%s)).\n\n",nextphase);
```

```
}

void fail_phase ()
{

  fprintf (out_file_ptr, "next_phase(%s)\n",phase_name);
  fprintf (out_file_ptr, "\t\t    :-    abort(%s),\n",phase_name);
  fprintf (out_file_ptr, "\t\t\t   retract(current_phase(");
  fprintf (out_file_ptr, "%s)),\n",phase_name);
  fprintf (out_file_ptr, "\t\t\t   asserta(current_phase(");
  fprintf (out_file_ptr, "%s)).\n\n\n",failphase);
}
```

This file is the controller.script file. It contains the mission independent code that is used for each mission. This file is copied to the new mission.pl.[descriptive name] and the mission dependent code is appended to it.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   Strategic Level Control Module controller.script
%
%   Written by Brad Leonhardt and Duane Davis
%
%   Modified last : 21 January 1996
%
%   Written using Quintus Prolog NOT COMPATIBLE with other prolog compilers
%
%   This program will control the Phoenix AUV through a series of waypoints,
%   gps fixes, depth changes, sonar searches, and hopefully back to starting
%   point.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                              %
%           Mission Controller Independent of Mission to be run                %
%                                                                              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%      Quintus specific to identify the .o file and functions within file.

foreign_file(tactical, [ood]).


%      Calls ood function which is a c function.  We send it a string and it
%      it returns an integer.

foreign(ood,c,ood(+string,[-integer])).


%      Program grabs tactical.o and math library (not sure if lm needed.)

                :-    load_foreign_files([tactical],
                          ['-lm -lsocket -lnsl']),
                          abolish(foreign_file/2),
                          abolish(foreign/3).


%      Conditions to exit program

done                 :-    current_phase(mission_abort),
                        ood('abort',X),X==1,
                           printsc('Abort mission').

done                 :-    current_phase(mission_complete),
                        ood('mission_complete',X),X==1,
                           printsc('Mission complete').

printsc(X)           :-    write(X),nl.
```

```
%       Loops through the Phases of mission

execute_mission     :-     initialize_mission,
                           repeat,execute_phase,done.


%       Prepare code and AUV for mission

initialize_mission :-      printsc('INITIALIZE'),
                        ood('initialize',X),X==1,
                        ood('start_timer 120',X),X==1,
                           repeat,initialized.

clean_up            :-     abolish(current_phase,1),
                           abolish(complete,1),
                           abolish(abort,1).

execute_phase       :-     current_phase(X),
                           execute_phase(X),
                           next_phase(X),!.

execute_phase (mission_abort).
next_phase (mission_abort).

execute_phase (mission_complete).
next_phase (mission_complete).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                      Start of Mission Dependent Code                       %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

initialized
                    :- ood('ask_time_out',X),X==1,
                           printsc('INITIALIZATION ABORTED DUE TO TIME OUT.'),
                           asserta(current_phase(mission_abort)).

initialized
                    :- ood('ask_initialized',X),X==1,
                           printsc('INITIALIZED.'),
                           clean_up,
% Mission Area Information for Moss Landing moss_info.pl

:- abolish(op_area/4), abolish(max_area_depth/1), abolish(area_depth/5),
   abolish(x_zero/1), abolish(y_zero/1), abolish(x_scale/1), abolish(y_scale/1).
:- path_clear(_,_).

:- asserta(op_area(-20,0,140,120)).

:- asserta(max_area_depth(20.0)).

:- asserta(area_depth(-20,110,90,140,-1)),
   asserta(area_depth(-20,100,80,110,-1)),
   asserta(area_depth(-20,90,60,100,-1)),
   asserta(area_depth(-20,80,40,90,-1)),
   asserta(area_depth(-20,70,20,80,-1)),
   asserta(area_depth(-20,60,0,70,-1)),
   asserta(area_depth(-20,50,-15,60,-1)),
   asserta(area_depth(-20,40,-15,50,-1)),
   asserta(area_depth(-20,30,-13,40,-1)),
   asserta(area_depth(-20,20,-10,30,-1)),
   asserta(area_depth(-20,10,-7,20,-1)),
   asserta(area_depth(-20,0,-2,10,-1)),
   asserta(area_depth(10,0,20,6,-1)),
   asserta(area_depth(20,0,30,10,-1)),
   asserta(area_depth(30,0,40,14,-1)),
   asserta(area_depth(40,0,50,19,-1)),
   asserta(area_depth(50,0,60,25,-1)),
   asserta(area_depth(60,0,70,27,-1)),
   asserta(area_depth(70,0,80,30,-1)),
   asserta(area_depth(80,0,90,36,-1)),
   asserta(area_depth(90,0,100,40,-1)),
   asserta(area_depth(100,0,110,45,-1)),
   asserta(area_depth(110,0,120,48,-1)),
   asserta(area_depth(120,0,130,52,-1)),
   asserta(area_depth(130,0,140,55,-1)).

:- asserta(x_zero(86)).
:- asserta(y_zero(558)).
:- asserta(x_scale(3.61)).
:- asserta(y_scale(-3.61)).
```

This is the mission.pl.mini_moss_landing code generated from the expert system. It is copied to mission.pl and then run inside Phoenix onboard the Voyager computer.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   Strategic Level Control Module
%
%   Written by Brad Leonhardt and Duane Davis
%
%   Modified last : 21 January 1996
%
%   Written using Quintus Prolog NOT COMPATIBLE with other prolog compilers
%
%   This program will control the Phoenix AUV through a series of waypoints,
%   gps fixes, depth changes, sonar searches, and hopefully back to starting
%   point.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                              %
%           Mission Controller Independent of Mission to be run               %
%                                                                              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%      Quintus specific to identify the .o file and functions within file.

foreign_file(tactical, [ood]).


%      Calls ood function which is a c function.  We send it a string and it
%      it returns an integer.

foreign(ood,c,ood(+string,[-integer])).


%      Program grabs tactical.o and math library (not sure if lm needed.)

                :-    load_foreign_files([tactical],
                          ['-lm -lsocket -lnsl']),
                          abolish(foreign_file/2),
                          abolish(foreign/3).


%      Conditions to exit program

done                 :-    current_phase(mission_abort),
                        ood('abort',X),X==1,
                           printsc('Abort mission').

done                 :-    current_phase(mission_complete),
                        ood('mission_complete',X),X==1,
                           printsc('Mission complete').

printsc(X)           :-    write(X),nl.
```

```
%       Loops through the Phases of mission

execute_mission     :-     initialize_mission,
                           repeat,execute_phase,done.


%       Prepare code and AUV for mission

initialize_mission :-      printsc('INITIALIZE'),
                        ood('initialize',X),X==1,
                        ood('start_timer 120',X),X==1,
                           repeat,initialized.

clean_up            :-     abolish(current_phase,1),
                           abolish(complete,1),
                           abolish(abort,1).

execute_phase       :-     current_phase(X),
                           execute_phase(X),
                           next_phase(X),!.

execute_phase (mission_abort).
next_phase (mission_abort).

execute_phase (mission_complete).
next_phase (mission_complete).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
                                                                     %%
                 Start of Mission Dependent Code                    %%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

initialized
                    :- ood('ask_time_out',X),X==1,
                           printsc('INITIALIZATION ABORTED DUE TO TIME OUT.'),
                           asserta(current_phase(mission_abort)).

initialized
                    :- ood('ask_initialized',X),X==1,
                           printsc('INITIALIZED.'),
                           clean_up,

                      asserta(current_phase(hover1)),
                      asserta(complete(0)),
                      asserta(abort(0)).

%       waypoint3


execute_phase(waypoint3)
                  :-    nl,printsc('PHASE waypoint3 STARTED.'),
                   ood('waypoint 10 60 3',X),X==1,
                      printsc('waypoint 10 60 3'),
                   ood('start_timer 500',X),X==1,
                      repeat,phase_completed(waypoint3).
```

```
phase_completed(waypoint3)
                    :- ood('ask_waypt_reached',X),X==1,
                        printsc('WAYPOINT REACHED.'),
                        asserta(complete(waypoint3)).

phase_completed(waypoint3)
                    :- ood('ask_time_out',X),X==1,
                        printsc('PHASE waypoint3 ABORTED DUE TO TIME OUT.'),
                        asserta(abort(waypoint3)).

next_phase(waypoint3)
                    :-    complete(waypoint3),
                        retract(current_phase(waypoint3)),
                        asserta(current_phase(hover6)).

next_phase(waypoint3)
                    :-    abort(waypoint3),
                        retract(current_phase(waypoint3)),
                        asserta(current_phase(mission_abort)).


%       hover6


execute_phase(hover6)
                    :-    nl,printsc('PHASE hover6 STARTED.'),
                     ood('hover 10 60 3 330',X),X==1,
                        printsc('hover 10 60 3 330'),
                     ood('start_timer 500',X),X==1,
                        repeat,phase_completed(hover6).

phase_completed(hover6)
                    :- ood('ask_hoverpt_reached',X),X==1,
                        printsc('HOVERPOINT REACHED.'),
                        asserta(complete(hover6)).

phase_completed(hover6)
                    :- ood('ask_time_out',X),X==1,
                        printsc('PHASE hover6 ABORTED DUE TO TIME OUT.'),
                        asserta(abort(hover6)).

next_phase(hover6)
                    :-    complete(hover6),
                        retract(current_phase(hover6)),
                        asserta(current_phase(hover7)).

next_phase(hover6)
                    :-    abort(hover6),
                        retract(current_phase(hover6)),
                        asserta(current_phase(mission_abort)).
```

```
%       hover16


execute_phase(hover16)
                  :-    nl,printsc('PHASE hover16 STARTED.'),
                   ood('hover 30 15 1 202',X),X==1,
                      printsc('hover 30 15 1 202'),
                   ood('start_timer 500',X),X==1,
                      repeat,phase_completed(hover16).

phase_completed(hover16)
                  :- ood('ask_hoverpt_reached',X),X==1,
                      printsc('HOVERPOINT REACHED.'),
                      asserta(complete(hover16)).

phase_completed(hover16)
                  :- ood('ask_time_out',X),X==1,
                      printsc('PHASE hover16 ABORTED DUE TO TIME OUT.'),
                      asserta(abort(hover16)).

next_phase(hover16)
                  :-    complete(hover16),
                      retract(current_phase(hover16)),
                      asserta(current_phase(mission_complete)).

next_phase(hover16)
                  :-    abort(hover16),
                      retract(current_phase(hover16)),
                      asserta(current_phase(mission_abort)).


%       gps_fix4


execute_phase(gps_fix4)
                  :-    nl,printsc('PHASE gps_fix4 STARTED.'),
                   ood('get_gps_fix',X),X==1,
                      printsc('get_gps_fix !'),
                   ood('start_timer 500',X),X==1,
                      repeat,phase_completed(gps_fix4).

phase_completed(gps_fix4)
                  :- ood('ask_get_gps_fix',X),X==1,
                      printsc('G P S FIX OBTAINED.'),
                      asserta(complete(gps_fix4)).

phase_completed(gps_fix4)
                  :- ood('ask_time_out',X),X==1,
                      printsc('PHASE gps_fix4 ABORTED DUE TO TIME OUT.'),
                      asserta(abort(gps_fix4)).

next_phase(gps_fix4)
                  :-    complete(gps_fix4),
                      retract(current_phase(gps_fix4)),
                      asserta(current_phase(hover16)).
```

```
next_phase(gps_fix4)
                    :-    abort(gps_fix4),
                        retract(current_phase(gps_fix4)),
                        asserta(current_phase(mission_abort)).


%       hover15


execute_phase(hover15)
                    :-    nl,printsc('PHASE hover15 STARTED.'),
                     ood('hover 30 15 3 202',X),X==1,
                        printsc('hover 30 15 3 202'),
                     ood('start_timer 500',X),X==1,
                        repeat,phase_completed(hover15).

phase_completed(hover15)
                    :- ood('ask_hoverpt_reached',X),X==1,
                        printsc('HOVERPOINT REACHED.'),
                        asserta(complete(hover15)).

phase_completed(hover15)
                    :- ood('ask_time_out',X),X==1,
                        printsc('PHASE hover15 ABORTED DUE TO TIME OUT.'),
                        asserta(abort(hover15)).

next_phase(hover15)
                    :-    complete(hover15),
                        retract(current_phase(hover15)),
                        asserta(current_phase(gps_fix4)).

next_phase(hover15)
                    :-    abort(hover15),
                        retract(current_phase(hover15)),
                        asserta(current_phase(mission_abort)).


%       hover14


execute_phase(hover14)
                    :-    nl,printsc('PHASE hover14 STARTED.'),
                     ood('hover 10 20 3 350',X),X==1,
                        printsc('hover 10 20 3 350'),
                     ood('start_timer 500',X),X==1,
                        repeat,phase_completed(hover14).

phase_completed(hover14)
                    :- ood('ask_hoverpt_reached',X),X==1,
                        printsc('HOVERPOINT REACHED.'),
                        asserta(complete(hover14)).

phase_completed(hover14)
                    :- ood('ask_time_out',X),X==1,
                        printsc('PHASE hover14 ABORTED DUE TO TIME OUT.'),
                        asserta(abort(hover14)).
```

```
next_phase(hover14)
                   :-    complete(hover14),
                       retract(current_phase(hover14)),
                       asserta(current_phase(hover15)).

next_phase(hover14)
                   :-    abort(hover14),
                       retract(current_phase(hover14)),
                       asserta(current_phase(mission_abort)).


%      waypoint6


execute_phase(waypoint6)
                   :-    nl,printsc('PHASE waypoint6 STARTED.'),
                    ood('waypoint 10 20 3',X),X==1,
                       printsc('waypoint 10 20 3'),
                    ood('start_timer 500',X),X==1,
                       repeat,phase_completed(waypoint6).

phase_completed(waypoint6)
                   :- ood('ask_waypt_reached',X),X==1,
                       printsc('WAYPOINT REACHED.'),
                       asserta(complete(waypoint6)).

phase_completed(waypoint6)
                   :- ood('ask_time_out',X),X==1,
                       printsc('PHASE waypoint6 ABORTED DUE TO TIME OUT.'),
                       asserta(abort(waypoint6)).

next_phase(waypoint6)
                   :-    complete(waypoint6),
                       retract(current_phase(waypoint6)),
                       asserta(current_phase(hover14)).

next_phase(waypoint6)
                   :-    abort(waypoint6),
                       retract(current_phase(waypoint6)),
                       asserta(current_phase(mission_abort)).


%      hover13


execute_phase(hover13)
                   :-    nl,printsc('PHASE hover13 STARTED.'),
                    ood('hover 0 40 3 295',X),X==1,
                       printsc('hover 0 40 3 295'),
                    ood('start_timer 500',X),X==1,
                       repeat,phase_completed(hover13).

phase_completed(hover13)
                   :- ood('ask_hoverpt_reached',X),X==1,
                       printsc('HOVERPOINT REACHED.'),
                       asserta(complete(hover13)).
```

```
phase_completed(hover13)
                  :- ood('ask_time_out',X),X==1,
                     printsc('PHASE hover13 ABORTED DUE TO TIME OUT.'),
                     asserta(abort(hover13)).

next_phase(hover13)
                  :-    complete(hover13),
                     retract(current_phase(hover13)),
                     asserta(current_phase(waypoint6)).

next_phase(hover13)
                  :-    abort(hover13),
                     retract(current_phase(hover13)),
                     asserta(current_phase(mission_abort)).


%       waypoint5


execute_phase(waypoint5)
                  :-    nl,printsc('PHASE waypoint5 STARTED.'),
                   ood('waypoint 0 40 3',X),X==1,
                      printsc('waypoint 0 40 3'),
                   ood('start_timer 500',X),X==1,
                      repeat,phase_completed(waypoint5).

phase_completed(waypoint5)
                  :- ood('ask_waypt_reached',X),X==1,
                     printsc('WAYPOINT REACHED.'),
                     asserta(complete(waypoint5)).

phase_completed(waypoint5)
                  :- ood('ask_time_out',X),X==1,
                     printsc('PHASE waypoint5 ABORTED DUE TO TIME OUT.'),
                     asserta(abort(waypoint5)).

next_phase(waypoint5)
                  :-    complete(waypoint5),
                     retract(current_phase(waypoint5)),
                     asserta(current_phase(hover13)).

next_phase(waypoint5)
                  :-    abort(waypoint5),
                     retract(current_phase(waypoint5)),
                     asserta(current_phase(mission_abort)).


%       hover12


execute_phase(hover12)
                  :-    nl,printsc('PHASE hover12 STARTED.'),
                   ood('hover 10 60 3 250',X),X==1,
                      printsc('hover 10 60 3 250'),
                   ood('start_timer 500',X),X==1,
                      repeat,phase_completed(hover12).
```

```
phase_completed(hover12)
                  :- ood('ask_hoverpt_reached',X),X==1,
                     printsc('HOVERPOINT REACHED.'),
                     asserta(complete(hover12)).

phase_completed(hover12)
                  :- ood('ask_time_out',X),X==1,
                     printsc('PHASE hover12 ABORTED DUE TO TIME OUT.'),
                     asserta(abort(hover12)).

next_phase(hover12)
                  :-   complete(hover12),
                     retract(current_phase(hover12)),
                     asserta(current_phase(waypoint5)).

next_phase(hover12)
                  :-   abort(hover12),
                     retract(current_phase(hover12)),
                     asserta(current_phase(mission_abort)).


%       waypoint4


execute_phase(waypoint4)
                  :-   nl,printsc('PHASE waypoint4 STARTED.'),
                   ood('waypoint 10 60 3',X),X==1,
                     printsc('waypoint 10 60 3'),
                   ood('start_timer 500',X),X==1,
                     repeat,phase_completed(waypoint4).

phase_completed(waypoint4)
                  :- ood('ask_waypt_reached',X),X==1,
                     printsc('WAYPOINT REACHED.'),
                     asserta(complete(waypoint4)).

phase_completed(waypoint4)
                  :- ood('ask_time_out',X),X==1,
                     printsc('PHASE waypoint4 ABORTED DUE TO TIME OUT.'),
                     asserta(abort(waypoint4)).

next_phase(waypoint4)
                  :-   complete(waypoint4),
                     retract(current_phase(waypoint4)),
                     asserta(current_phase(hover12)).

next_phase(waypoint4)
                  :-   abort(waypoint4),
                     retract(current_phase(waypoint4)),
                     asserta(current_phase(mission_abort)).
```

```
%       hover11

execute_phase(hover11)
                    :-    nl,printsc('PHASE hover11 STARTED.'),
                     ood('hover 80 70 3 190',X),X==1,
                        printsc('hover 80 70 3 190'),
                     ood('start_timer 500',X),X==1,
                        repeat,phase_completed(hover11).

phase_completed(hover11)
                    :- ood('ask_hoverpt_reached',X),X==1,
                        printsc('HOVERPOINT REACHED.'),
                        asserta(complete(hover11)).

phase_completed(hover11)
                    :- ood('ask_time_out',X),X==1,
                        printsc('PHASE hover11 ABORTED DUE TO TIME OUT.'),
                        asserta(abort(hover11)).

next_phase(hover11)
                    :-    complete(hover11),
                        retract(current_phase(hover11)),
                        asserta(current_phase(waypoint4)).

next_phase(hover11)
                    :-    abort(hover11),
                        retract(current_phase(hover11)),
                        asserta(current_phase(mission_abort)).


%       gps_fix3


execute_phase(gps_fix3)
                    :-    nl,printsc('PHASE gps_fix3 STARTED.'),
                     ood('get_gps_fix',X),X==1,
                        printsc('get_gps_fix !'),
                     ood('start_timer 500',X),X==1,
                        repeat,phase_completed(gps_fix3).

phase_completed(gps_fix3)
                    :- ood('ask_get_gps_fix',X),X==1,
                        printsc('G P S FIX OBTAINED.'),
                        asserta(complete(gps_fix3)).

phase_completed(gps_fix3)
                    :- ood('ask_time_out',X),X==1,
                        printsc('PHASE gps_fix3 ABORTED DUE TO TIME OUT.'),
                        asserta(abort(gps_fix3)).

next_phase(gps_fix3)
                    :-    complete(gps_fix3),
                        retract(current_phase(gps_fix3)),
                        asserta(current_phase(hover11)).
```

```
next_phase(gps_fix3)
                    :-    abort(gps_fix3),
                        retract(current_phase(gps_fix3)),
                        asserta(current_phase(mission_abort)).


%       hover10


execute_phase(hover10)
                    :-    nl,printsc('PHASE hover10 STARTED.'),
                     ood('hover 80 70 3 0',X),X==1,
                        printsc('hover 80 70 3 0'),
                     ood('start_timer 500',X),X==1,
                        repeat,phase_completed(hover10).

phase_completed(hover10)
                    :- ood('ask_hoverpt_reached',X),X==1,
                        printsc('HOVERPOINT REACHED.'),
                        asserta(complete(hover10)).

phase_completed(hover10)
                    :- ood('ask_time_out',X),X==1,
                        printsc('PHASE hover10 ABORTED DUE TO TIME OUT.'),
                        asserta(abort(hover10)).

next_phase(hover10)
                    :-    complete(hover10),
                        retract(current_phase(hover10)),
                        asserta(current_phase(gps_fix3)).

next_phase(hover10)
                    :-    abort(hover10),
                        retract(current_phase(hover10)),
                        asserta(current_phase(mission_abort)).


%       rotate_search


execute_phase(rotate_search)
                    :-    nl,printsc('PHASE rotate_search STARTED.'),
                     ood('sonar_search',X),X==1,
                        printsc('sonar_search !'),
                     ood('start_timer 500',X),X==1,
                        repeat,phase_completed(rotate_search).

phase_completed(rotate_search)
                    :- ood('ask_sonar_search_complete',X),X==1,
                        printsc('SONAR SEARCH COMPLETE.'),
                        asserta(complete(rotate_search)).

phase_completed(rotate_search)
                    :- ood('ask_time_out',X),X==1,
                        printsc('PHASE rotate_search ABORTED DUE TO TIME OUT.'),
                        asserta(abort(rotate_search)).
```

```
next_phase(rotate_search)
                    :-    complete(rotate_search),
                        retract(current_phase(rotate_search)),
                        asserta(current_phase(hover10)).

next_phase(rotate_search)
                    :-    abort(rotate_search),
                        retract(current_phase(rotate_search)),
                        asserta(current_phase(mission_abort)).


%       rotate_sonar


execute_phase(rotate_sonar)
                    :-    nl,printsc('PHASE rotate_sonar STARTED.'),
                     ood('rotate_search',X),X==1,
                        printsc('rotate_search !'),
                     ood('start_timer 500',X),X==1,
                        repeat,phase_completed(rotate_sonar).

phase_completed(rotate_sonar)
                    :- ood('ask_rotate_search_complete',X),X==1,
                        printsc('ROTATE SEARCH COMPLETE.'),
                        asserta(complete(rotate_sonar)).

phase_completed(rotate_sonar)
                    :- ood('ask_time_out',X),X==1,
                        printsc('PHASE rotate_sonar ABORTED TIME OUT.'),
                        asserta(abort(rotate_sonar)).

next_phase(rotate_sonar)
                    :-    complete(rotate_sonar),
                        retract(current_phase(rotate_sonar)),
                        asserta(current_phase(gps_fix2)).

next_phase(rotate_sonar)
                    :-    abort(rotate_sonar),
                        retract(current_phase(rotate_sonar)),
                        asserta(current_phase(mission_abort)).


%       hover9


execute_phase(hover9)
                    :-    nl,printsc('PHASE hover9 STARTED.'),
                     ood('hover 80 70 3 0',X),X==1,
                        printsc('hover 80 70 3 0'),
                     ood('start_timer 500',X),X==1,
                        repeat,phase_completed(hover9).

phase_completed(hover9)
                    :- ood('ask_hoverpt_reached',X),X==1,
                        printsc('HOVERPOINT REACHED.'),
                        asserta(complete(hover9)).
```

```
phase_completed(hover9)
                    :- ood('ask_time_out',X),X==1,
                         printsc('PHASE hover9 ABORTED DUE TO TIME OUT.'),
                         asserta(abort(hover9)).

next_phase(hover9)
                    :-     complete(hover9),
                         retract(current_phase(hover9)),
                         asserta(current_phase(rotate_search)).

next_phase(hover9)
                    :-     abort(hover9),
                         retract(current_phase(hover9)),
                         asserta(current_phase(mission_abort)).


%       hover8


execute_phase(hover8)
                    :-     nl,printsc('PHASE hover8 STARTED.'),
                      ood('hover 30 50 3 20',X),X==1,
                         printsc('hover 30 50 3 20'),
                      ood('start_timer 500',X),X==1,
                         repeat,phase_completed(hover8).

phase_completed(hover8)
                    :- ood('ask_hoverpt_reached',X),X==1,
                         printsc('HOVERPOINT REACHED.'),
                         asserta(complete(hover8)).

phase_completed(hover8)
                    :- ood('ask_time_out',X),X==1,
                         printsc('PHASE hover8 ABORTED DUE TO TIME OUT.'),
                         asserta(abort(hover8)).

next_phase(hover8)
                    :-     complete(hover8),
                         retract(current_phase(hover8)),
                         asserta(current_phase(hover9)).

next_phase(hover8)
                    :-     abort(hover8),
                         retract(current_phase(hover8)),
                         asserta(current_phase(mission_abort)).


%       gps_fix2


execute_phase(gps_fix2)
                    :-     nl,printsc('PHASE gps_fix2 STARTED.'),
                      ood('get_gps_fix',X),X==1,
                         printsc('get_gps_fix !'),
                      ood('start_timer 500',X),X==1,
                         repeat,phase_completed(gps_fix2).
```

```
phase_completed(gps_fix2)
                 :- ood('ask_get_gps_fix',X),X==1,
                    printsc('G P S FIX OBTAINED.'),
                    asserta(complete(gps_fix2)).

phase_completed(gps_fix2)
                 :- ood('ask_time_out',X),X==1,
                    printsc('PHASE gps_fix2 ABORTED DUE TO TIME OUT.'),
                    asserta(abort(gps_fix2)).

next_phase(gps_fix2)
                 :-    complete(gps_fix2),
                    retract(current_phase(gps_fix2)),
                    asserta(current_phase(hover8)).

next_phase(gps_fix2)
                 :-    abort(gps_fix2),
                    retract(current_phase(gps_fix2)),
                    asserta(current_phase(mission_abort)).


%       hover7


execute_phase(hover7)
                 :-    nl,printsc('PHASE hover7 STARTED.'),
                  ood('hover 30 50 3 20',X),X==1,
                     printsc('hover 30 50 3 20'),
                  ood('start_timer 500',X),X==1,
                     repeat,phase_completed(hover7).

phase_completed(hover7)
                 :- ood('ask_hoverpt_reached',X),X==1,
                    printsc('HOVERPOINT REACHED.'),
                    asserta(complete(hover7)).

phase_completed(hover7)
                 :- ood('ask_time_out',X),X==1,
                    printsc('PHASE hover7 ABORTED DUE TO TIME OUT.'),
                    asserta(abort(hover7)).

next_phase(hover7)
                 :-    complete(hover7),
                    retract(current_phase(hover7)),
                    asserta(current_phase(rotate_sonar)).

next_phase(hover7)
                 :-    abort(hover7),
                    retract(current_phase(hover7)),
                    asserta(current_phase(mission_abort)).
```

```
%       hover5


execute_phase(hover5)
                  :-    nl,printsc('PHASE hover5 STARTED.'),
                   ood('hover 0 40 3 70',X),X==1,
                      printsc('hover 0 40 3 70'),
                   ood('start_timer 500',X),X==1,
                      repeat,phase_completed(hover5).

phase_completed(hover5)
                  :- ood('ask_hoverpt_reached',X),X==1,
                      printsc('HOVERPOINT REACHED.'),
                      asserta(complete(hover5)).

phase_completed(hover5)
                  :- ood('ask_time_out',X),X==1,
                      printsc('PHASE hover5 ABORTED DUE TO TIME OUT.'),
                      asserta(abort(hover5)).

next_phase(hover5)
                  :-    complete(hover5),
                      retract(current_phase(hover5)),
                      asserta(current_phase(waypoint3)).

next_phase(hover5)
                  :-    abort(hover5),
                      retract(current_phase(hover5)),
                      asserta(current_phase(mission_abort)).


%       hover4


execute_phase(hover4)
                  :-    nl,printsc('PHASE hover4 STARTED.'),
                   ood('hover 10 20 3 115',X),X==1,
                      printsc('hover 10 20 3 115'),
                   ood('start_timer 500',X),X==1,
                      repeat,phase_completed(hover4).

phase_completed(hover4)
                  :- ood('ask_hoverpt_reached',X),X==1,
                      printsc('HOVERPOINT REACHED.'),
                      asserta(complete(hover4)).

phase_completed(hover4)
                  :- ood('ask_time_out',X),X==1,
                      printsc('PHASE hover4 ABORTED DUE TO TIME OUT.'),
                      asserta(abort(hover4)).

next_phase(hover4)
                  :-    complete(hover4),
                      retract(current_phase(hover4)),
                      asserta(current_phase(waypoint2)).
```

```
next_phase(hover4)
                    :-    abort(hover4),
                        retract(current_phase(hover4)),
                        asserta(current_phase(mission_abort)).


%       waypoint1


execute_phase(waypoint1)
                    :-    nl,printsc('PHASE waypoint1 STARTED.'),
                     ood('waypoint 10 20 3',X),X==1,
                        printsc('waypoint 10 20 3'),
                     ood('start_timer 500',X),X==1,
                        repeat,phase_completed(waypoint1).

phase_completed(waypoint1)
                    :- ood('ask_waypt_reached',X),X==1,
                        printsc('WAYPOINT REACHED.'),
                        asserta(complete(waypoint1)).

phase_completed(waypoint1)
                    :- ood('ask_time_out',X),X==1,
                        printsc('PHASE waypoint1 ABORTED DUE TO TIME OUT.'),
                        asserta(abort(waypoint1)).

next_phase(waypoint1)
                    :-    complete(waypoint1),
                        retract(current_phase(waypoint1)),
                        asserta(current_phase(hover4)).

next_phase(waypoint1)
                    :-    abort(waypoint1),
                        retract(current_phase(waypoint1)),
                        asserta(current_phase(mission_abort)).


%       waypoint2


execute_phase(waypoint2)
                    :-    nl,printsc('PHASE waypoint2 STARTED.'),
                     ood('waypoint 0 40 3',X),X==1,
                        printsc('waypoint 0 40 3'),
                     ood('start_timer 500',X),X==1,
                        repeat,phase_completed(waypoint2).

phase_completed(waypoint2)
                    :- ood('ask_waypt_reached',X),X==1,
                        printsc('WAYPOINT REACHED.'),
                        asserta(complete(waypoint2)).

phase_completed(waypoint2)
                    :- ood('ask_time_out',X),X==1,
                        printsc('PHASE waypoint2 ABORTED DUE TO TIME OUT.'),
                        asserta(abort(waypoint2)).
```

```
next_phase(waypoint2)
                    :-    complete(waypoint2),
                        retract(current_phase(waypoint2)),
                        asserta(current_phase(hover5)).

next_phase(waypoint2)
                    :-    abort(waypoint2),
                        retract(current_phase(waypoint2)),
                        asserta(current_phase(mission_abort)).


%       hover3


execute_phase(hover3)
                    :-    nl,printsc('PHASE hover3 STARTED.'),
                     ood('hover 30 15 3 170',X),X==1,
                        printsc('hover 30 15 3 170'),
                     ood('start_timer 500',X),X==1,
                        repeat,phase_completed(hover3).

phase_completed(hover3)
                    :- ood('ask_hoverpt_reached',X),X==1,
                        printsc('HOVERPOINT REACHED.'),
                        asserta(complete(hover3)).

phase_completed(hover3)
                    :- ood('ask_time_out',X),X==1,
                        printsc('PHASE hover3 ABORTED DUE TO TIME OUT.'),
                        asserta(abort(hover3)).

next_phase(hover3)
                    :-    complete(hover3),
                        retract(current_phase(hover3)),
                        asserta(current_phase(waypoint1)).

next_phase(hover3)
                    :-    abort(hover3),
                        retract(current_phase(hover3)),
                        asserta(current_phase(mission_abort)).


%       hover2


execute_phase(hover2)
                    :-    nl,printsc('PHASE hover2 STARTED.'),
                     ood('hover 30 15 3 202',X),X==1,
                        printsc('hover 30 15 3 202'),
                     ood('start_timer 500',X),X==1,
                        repeat,phase_completed(hover2).

phase_completed(hover2)
                    :- ood('ask_hoverpt_reached',X),X==1,
                        printsc('HOVERPOINT REACHED.'),
                        asserta(complete(hover2)).
```

```
phase_completed(hover2)
                  :- ood('ask_time_out',X),X==1,
                     printsc('PHASE hover2 ABORTED DUE TO TIME OUT.'),
                     asserta(abort(hover2)).

next_phase(hover2)
                  :-    complete(hover2),
                     retract(current_phase(hover2)),
                     asserta(current_phase(hover3)).

next_phase(hover2)
                  :-    abort(hover2),
                     retract(current_phase(hover2)),
                     asserta(current_phase(mission_abort)).


%       gps_fix1


execute_phase(gps_fix1)
                  :-    nl,printsc('PHASE gps_fix1 STARTED.'),
                   ood('get_gps_fix',X),X==1,
                      printsc('get_gps_fix !'),
                   ood('start_timer 500',X),X==1,
                      repeat,phase_completed(gps_fix1).

phase_completed(gps_fix1)
                  :- ood('ask_get_gps_fix',X),X==1,
                     printsc('G P S FIX OBTAINED.'),
                     asserta(complete(gps_fix1)).

phase_completed(gps_fix1)
                  :- ood('ask_time_out',X),X==1,
                     printsc('PHASE gps_fix1 ABORTED DUE TO TIME OUT.'),
                     asserta(abort(gps_fix1)).

next_phase(gps_fix1)
                  :-    complete(gps_fix1),
                     retract(current_phase(gps_fix1)),
                     asserta(current_phase(hover2)).

next_phase(gps_fix1)
                  :-    abort(gps_fix1),
                     retract(current_phase(gps_fix1)),
                     asserta(current_phase(mission_abort)).


%       hover1


execute_phase(hover1)
                  :-    nl,printsc('PHASE hover1 STARTED.'),
                   ood('hover 30 15 1 202',X),X==1,
                      printsc('hover 30 15 1 202'),
                   ood('start_timer 500',X),X==1,
                      repeat,phase_completed(hover1).
```

```
phase_completed(hover1)
                 :- ood('ask_hoverpt_reached',X),X==1,
                    printsc('HOVERPOINT REACHED.'),
                    asserta(complete(hover1)).

phase_completed(hover1)
                 :- ood('ask_time_out',X),X==1,
                    printsc('PHASE hover1 ABORTED DUE TO TIME OUT.'),
                    asserta(abort(hover1)).

next_phase(hover1)
                 :-    complete(hover1),
                    retract(current_phase(hover1)),
                    asserta(current_phase(gps_fix1)).

next_phase(hover1)
                 :-    abort(hover1),
                    retract(current_phase(hover1)),
                    asserta(current_phase(mission_abort)).
```

APPENDIX D. VXWORKS

A. INTRODUCTION

The following report is a brief summary of VxWorks Programmer Guide, Release 5.1, [Wind River Systems 93]. The other major source of information came from the "comp.os.vxworks" newsgroup. Multitasking and interprocess communication are complementary concepts. Several tasks can run at the same time and exchange data with each other. This is the prerequisite of a real-time system, because the real world consists of multiple and discrete events. VxWorks provides a large range of intertask communication facilities, such as semaphores, message queues, pipes and network-transparent sockets. The following discussion describes these processes and tools.

VxWorks provides a real-time kernel which interleaves the execution of multiple tasks employing a scheduling algorithm. Thus the user sees multiple tasks executing simultaneously. To avoid virtual-to-physical memory mapping, VxWorks uses a single common address space for all tasks. However, virtual-to-physical memory mapping is available through an additional product. The VxWorks kernel uses four states to manage the tasks, suspended, delayed, ready and pended. This is different from [Tanenbaum 92], who describes only three states, running, blocked and ready.

In VxWorks, when a process is created, it enters the suspended state. It must be activated to enter the ready state and thus be available for execution. A primitive is supplied for both creating and activating a task. This primitive is referred to as spawning. The suspended state is primarily used for debugging. Tasks in the ready state are not waiting for any resource other than the CPU. This means they are runnable and only temporarily stopped to let another process run. In the pended state there are no resources available for a task and the task is blocked. A task in the delayed state is sleeping for some duration. Combinations of states are also possible. A task can be both delayed and suspended, pended and suspended, pended with a timeout value, and pended and suspended with a time out value. Tasks can be deleted from any state.

Preemptive priority scheduling is the default scheduling algorithm in VxWorks. Each task is assigned a priority, and the kernel ensures that the task with the highest priority will have the CPU. The preemption will cause the running task to be pended immediately whenever a task with a higher priority becomes ready to run. VxWorks provides 256 priority levels, 0 being the highest, and 255 the lowest. When a task is created, a priority is assigned. However, the priority may change during task execution (dynamic prioritization). This is an important feature when real-world matters come into play. If necessary, round-robin scheduling can be enabled. This ensures that all tasks of the same priority can share the CPU equally, thus avoiding the scenario of one task usurping the CPU longer than other tasks of the same priority. To achieve this, a run-time counter is implemented. When the specified time interval is over, the task is placed at the tail of the task queue of its priority. New tasks also join the queue at the tail. Tasks can be programmed to explicitly disable or enable the VxWorks scheduler. In the case of a tasklock (i.e., disable), priority-based preemption will not occur. Should the task block itself or become suspended, the scheduler will select the next task with the highest priority. This technique can be used to achieve mutual exclusion.

The libraries taskLib and usrLib provide routines for task implementation. For example, the routine taskSpawn() is used to create and activate a task (Figure 43).

```
id = taskSpawn(name, priority, options, stacksize, main, arg1, ..., arg10.
```

Figure 43. Spawning a task in VxWorks.

The task name is an ASCII string of any length, preferably starting with the letter t to avoid name conflicts. An example of how the main routine is setup is shown in (Figure 44).

In VxWorks, several different tasks may invoke a single copy of a subroutine. An example is the printf() function call. A single copy of code which is executed by multiple tasks is called shared code. By use of dynamic linking facilities VxWorks makes this task simple. Shared code must be reentrant. To be reentrant, a subroutine must be capable of being called from several task contexts simultaneously without conflict. Conflicts with reentrant code occur when global or static variables are changed, since there is only one copy of the code. VxWorks makes shared code reentrant by use of dynamic stack variables guarding of global and static variables with semaphores use of task variables.

```
taskOne(void)
{
        ...;
        ...;
        myFunc();
        ...;
}
```

Figure 44. Sample task function.

Most subroutines perform work on data that is supplied to them. They do not have data of their own. This pure code utilizes dynamic stack variables. Each time the subroutine is called, a new stack is created thus preventing interference between multiple tasks. Some VxWorks libraries contain access to common data. In order to prohibit simultaneously executing critical sections of code a mutual-exclusion mechanism is required. This is accomplished through the use of semaphores.

Some routines which may be called by several tasks at the same time may need global or static variables which have a distinct value for each of the calling tasks. By use of 4-byte variables added to a task's context, called task variables, this can be accommodated. This facility should be used sparingly as it adds a few microseconds to the context switching process. It is possible, with VxWorks, to spawn several tasks with the same main routine. Since each spawn creates a new task with its own stack and context, it can also pass the main routine different parameters to the new task. Due to multitasking , there must be a means to permit coordination of actions of independent tasks.

Some of the intertask communications mechanisms employed are:

- shared memory, for simple sharing of data
- semaphores, for basic mutual exclusion and synchronization
- message queues and pipes, for intertask message passing within a CPU
- signals, for exception handling.
- sockets and remote procedure calls (RPC's), for network-transparent intertask communication

By means of shared data structures, tasks can communicate. All tasks in VxWorks exist in a single linear address space. Global variables, linear buffers, ring buffers, linked lists, and pointers can be referenced directly by code that is running in different contexts. Mutual Exclusion can be accomplished through several methods. Disabling interrupts is the most powerful method available. It will guarantee exclusive access to the CPU. However, it is inappropriate for general purpose use as it disables the system from responding to external events for the duration of these locks. Preemptive locks are a less restrictive form of mutual exclusion. They prevent other tasks from preempting the current executing task, but allow interrupt service routines to execute. The downside to this setup is that tasks of higher priority are not able to execute until the locking task exits the critical region, even if the higher priority task is not involved with the critical region.

Semaphores are the primary means for addressing the requirements of both mutual exclusion and task synchronization. In mutual exclusion, VxWorks semaphores interlock access to shared resources. Finer granularity than either interrupt disabling or preemptive locks is obtained. In synchronization, VxWorks semaphores coordinate a task's execution with some external event.

Modern real-time applications are constructed as a set of independent but cooperating tasks. Semaphores provide a high-speed mechanism for the synchronization and interlocking of tasks, but often a higher-level mechanism is needed to allow cooperation tasks to communicate with each other. The primary intertask communication mechanism within a single CPU is message queues. VxMP is a global message queue that can be used across processors. Message queues allow messages of variable length to be queued in FIFO order. Any task or interrupt service routine can send messages to a message queue. Full-duplex communication between two tasks generally requires two message queues, one for each direction.

Real-time systems are often structured using a client-server model of tasks. The server tasks accept requests from the client tasks to perform a service, and then usually return a reply. To send the request or reply intertask messages are used, which in VxWorks normally means message queues or pipes. Pipes provide another interface to the message queue facility that goes through the VxWorks I/O system. Pipes are virtual I/O devices managed by the driver pipeDrv. The function, pipeDevCreate() creates a pipe device and the associated message queue. Tasks block when they read from an empty pipe until data is available, and block when they write to a full pipe until there is space available. Like message queues, interrupt service routines can write to a pipe, but cannot read from a pipe. Pipes provide an important feature that message queues do not. The select() routine allows a task to wait for data to be available on I/O devices. The select() routine also works with other asynchronous I/O devices including network sockets and serial devices.

VxWorks uses sockets as the basis of intertask communications across the network. A socket is an endpoint for communications between tasks; that is, data is sent from one socket to another. The internet communications protocols that VxWorks support are TCP and UDP. TCP provides two-way guaranteed transmission of data via stream sockets. The two sockets are "connected" providing for a reliable byte-stream to flow in each direction. UDP is simpler and less robust than TCP. It sends separate unconnected individually addressed packets called datagrams. What makes socket communications so good is that communications among processes are exactly the same regardless of the location of the processes in the network or the operating system under which they are running.

RPC is a facility that allows a process on one machine to call a procedure which is executed by another process on either the same machine or a remote machine. RPC uses sockets for the communications. This means VxWorks tasks and UNIX processes can invoke routines that are executed on other VxWorks or UNIX machines, in any combination.

VxWorks supports a software signaling facility. Signals asynchronously alter the control flow of a task. Any task or interrupt service routine can raise a signal for a particular task. The signaled task immediately suspends its current thread of execution and the task-specified signal handler routine is executed the next time the task is scheduled to run.

Hardware interrupts are important in real-time systems, because they let the system quickly respond to external events. VxWorks provides a special context for interrupt service code to avoid task context switching, and thus renders fast response. VxWorks supplies interrupt routines which connect to C functions and pass arguments to the functions to be executed at interrupt level. To return from an interrupt, the connected function simply returns. A routine connected to an interrupt in this way is referred to as an interrupt service routine (ISR) or interrupt handler. When an interrupt occurs, the registers are saved, a stack for the arguments to be passed is set up, then the C function is called. On return, stack and registers are restored.

Interrupt service code does not run in a regular task context. This is why ISRs may not call functions which could cause blocking of the caller. They may give a semaphore, however, but may not take one. Also, VxWorks drivers may not be used, because most of them require a task context. For the user the VxWorks I/O system looks like the UNIX I/O system. It provides an interface to the following devices:

- ♦ character-oriented devices such as terminals and communication lines
- ♦ random-access block devices such as disks
- ♦ virtual devices such as intertask pipes and sockets
- ♦ monitor and control devices such as digital/analog I/O devices
- ♦ network devices that give access to remote devices

  and others.

Internally, however, the VxWorks I/O system is uniquely designed to make it faster than a UNIX system to comply to real-time system needs. Like in UNIX, all I/O in VxWorks is directed at named files. File names can refer to two things:

- ♦ unstructured "raw" devices such as a serial communications channel or an intertask pipe
- ♦ a logical file on a random access device such as a disk.

However, the devices are handled by device drivers which are special program modules. There are two levels for I/O in VxWorks: basic and buffered. They are different not only in the way data is buffered, but also in the types of calls that can be applied.

Basic I/O is the lowest I/O level in VxWorks. Basic primitives like open(), close(), read(), and write() are provided in C libraries. When a file is opened, an integer file descriptor (fd) is assigned and subsequently used as a parameter to address the file. The three standard devices, standard input, standard output, and standard error output, possess reserved fd's (0, 1, 2). Only a limited number of fd's is available. Thus it is important to close a file when it is no longer needed. When a file is closed, all I/O to this file is completely written out (flushed). It is important to note that fd's are not task specific. When a file is opened by a task, it will not be automatically closed when the task exits. This must be explicitly programmed. A select()-facility gives tasks the ability to wait for multiple devices simultaneously and specify the maximum time to wait. For example, if a server task uses a pipe to communicate with local clients and a socket to communicate with remote clients, the server normally blocks on one device, waiting for a request, not being able to service requests on the other. With the select()-facility the server can monitor both devices and respond quickly.

Buffered I/O is the standard I/O method in VxWorks. To avoid the repeated overhead of calls to the I/O system when a file is processed a character at a time, a buffer facility is provided. Data is processed in large chunks and stored in a buffer transparent to the application. The system assigns a file pointer (fp) to each device, contrary to a file descriptor in basic I/O, which is simply an integer. The file pointer is a pointer to the associated data structure of type FILE, a handle for the opened file and its associated buffers and pointers. The FILE structure contains a file descriptor. When, for example, a read() is issued, the system reads data from the file until the buffer is full. Then the pointer to the buffer is advanced one character at a time to simulate a character-at-a-time read. When the buffer is empty, a new data chunk is read from the file. The efficiency of this process is obvious. Most general uses of I/O in VxWorks are completely source-compatible with I/O in UNIX. However, the following differences are noted:

- ♦ Device Configuration. In VxWorks, device drivers can be installed and removed dynamically.
- ♦ File Descriptors. In UNIX, fds are unique to each process. In VxWorks, fds are global entities, accessible by any task, except for standard input, standard output, and standard error (0, 1, 2), which can be task-specific. I/O Control. The specific parameters passed in ioctrl() functions may be different in UNIX and VxWorks.
- ♦ Driver Routines. In UNIX, device drivers execute in system mode and are not preemptible. In VxWorks, driver routines are in fact preemptible since they execute within the context of the task that invoked them.

In many systems, device drivers supply a few functions to perform low-level I/O functions; i.e., inputting or outputting a sequence of bytes to character-oriented devices. Higher level protocols; i.e., for communications, are implemented in the device-independent part of the I/O system. User requests are submitted to the I/O system, where they are processed, and then the driver functions take over. This makes it easy to implement the drivers, but programming becomes more difficult when alternative protocols that are not provided by the I/O system shall be written. In real-time systems it may be preferable to bypass standard protocols in order to gain performance. In VxWorks, the I/O system simply acts like a switch to route the user requests to the appropriate driver-supplied routines. The I/O system performs little processing. The drivers can process the raw user requests as appropriate to its devices. In addition to this, standard protocols are available to the programmer in high-level subroutine libraries. VxWorks supplies two fundamental types of devices: block devices with random access which can be used for storing file systems and where data is transferred in blocks (i.e., floppy and hard disks), and character devices which are defined to be everything else but block devices (i.e., serial and graphical input devices such as terminals and graphics tablets).

B. SUMMARY

VxWorks is a development and execution environment for complex real-time applications supported on a wide variety of processors. The three major components of the VxWorks system are: a high performance scalable real-time operating system; a cross-development tool used on a host development system; and a wide range of communications software options such as Ethernet for connection of the host to the target.

APPENDIX E. OBTAINING AND RUNNING CURRENT SOFTWARE

One of the Naval Postgraduate School's primary missions is to provide information for the military and public. The Center for Autonomous Vehicle Research (CAUVR) (Figure 45) makes all its significant work available almost immediately to anyone around the world. Via the Internet, copies of all current software which is used to run the virtual world simulator or Phoenix are available for downloading. Other items available include: numerous graphics, photographs, theses, dissertations, papers, briefings, daily meetings, personnel listings, relating to AUV research at NPS (Figure 46).

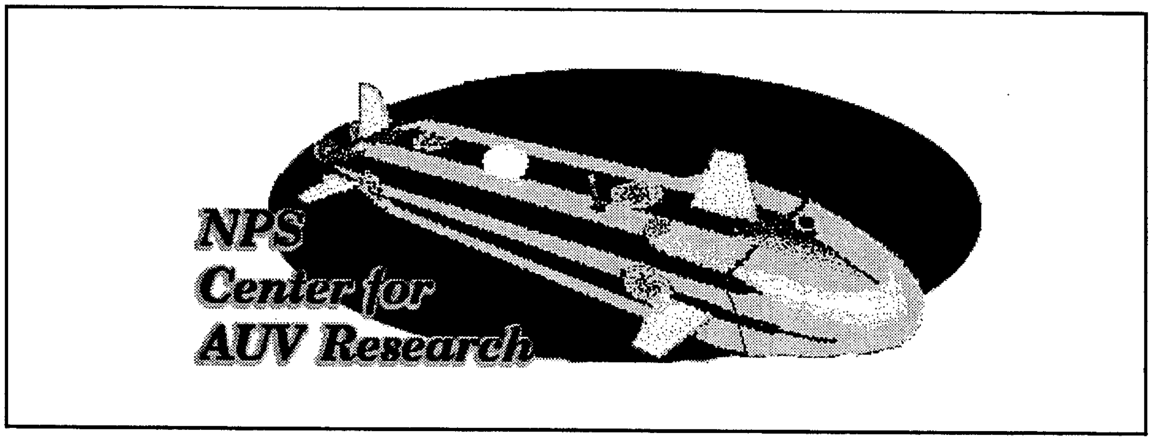
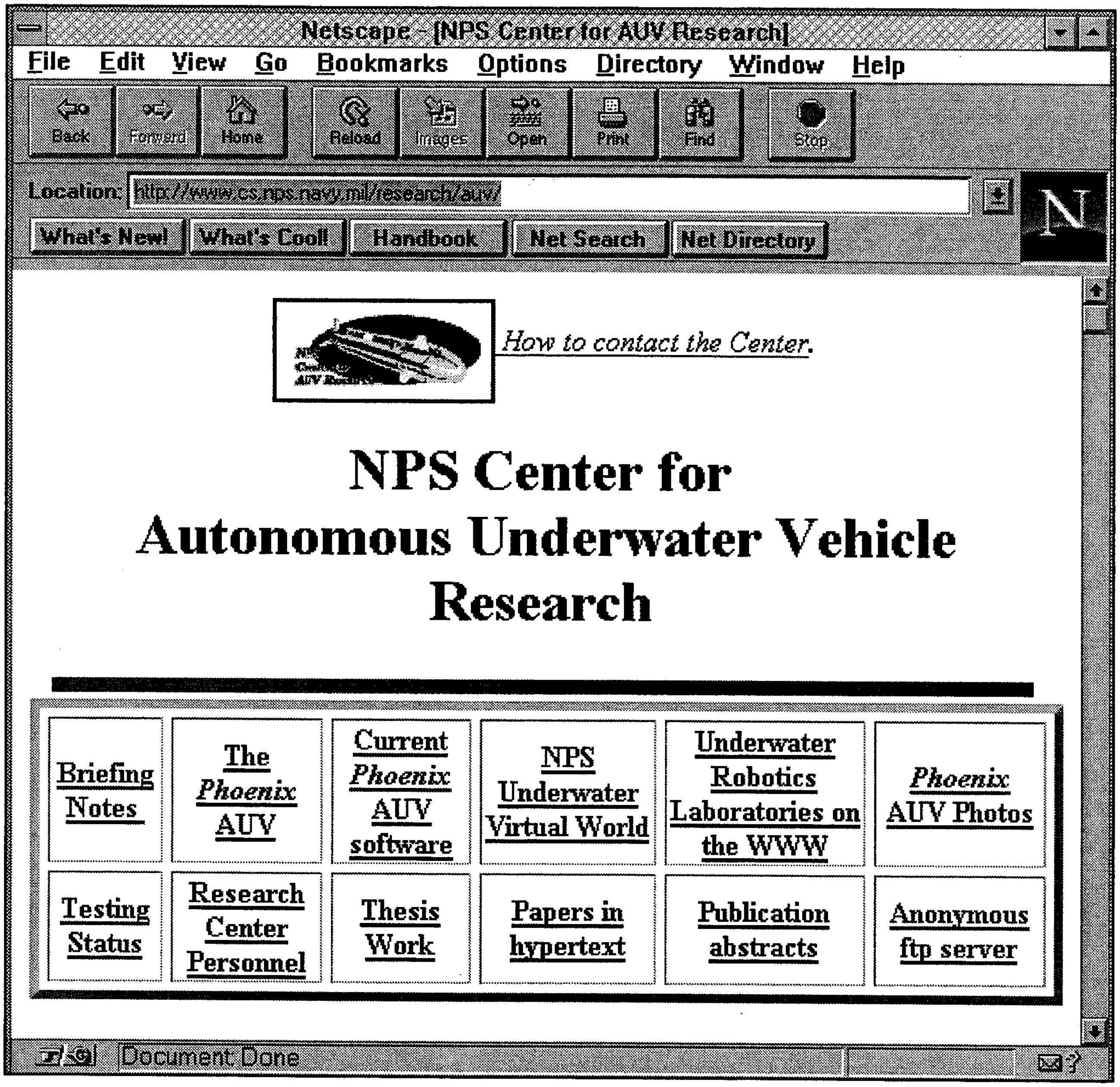


Figure 45. CAUVR logo.

An email group (auvrg@cs.nps.navy.mil) has been created to rapidly send message traffic to all members involved in the research group. The email group can be subscribed to by filling out a request, which is available through CAUVR web site.


Netscape - [NPS Center for AUV Research]
File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Reload Images Open Print Find Stop
Location: http://www.cs.nps.navy.mil/research/auv/
What's New! What's Cool! Handbook Net Search Net Directory
Document: Done

How to contact the Center.

NPS Center for
Autonomous Underwater Vehicle
Research

Briefing Notes	The *Phoenix* AUV	Current *Phoenix* AUV software	NPS Underwater Virtual World	Underwater Robotics Laboratories on the WWW	*Phoenix* AUV Photos
Testing Status	Research Center Personnel	Thesis Work	Papers in hypertext	Publication abstracts	Anonymous ftp server

Figure 46. CAUVR web site is: http://www.cs.nps.navy.mil/research/auv.

Files for the software can be downloaded individually or a complete download of the complete software package is available. The complete download and installation instructions are available at the Software Reference site. The size of the complete downloaded zip file is around 15 Mb.

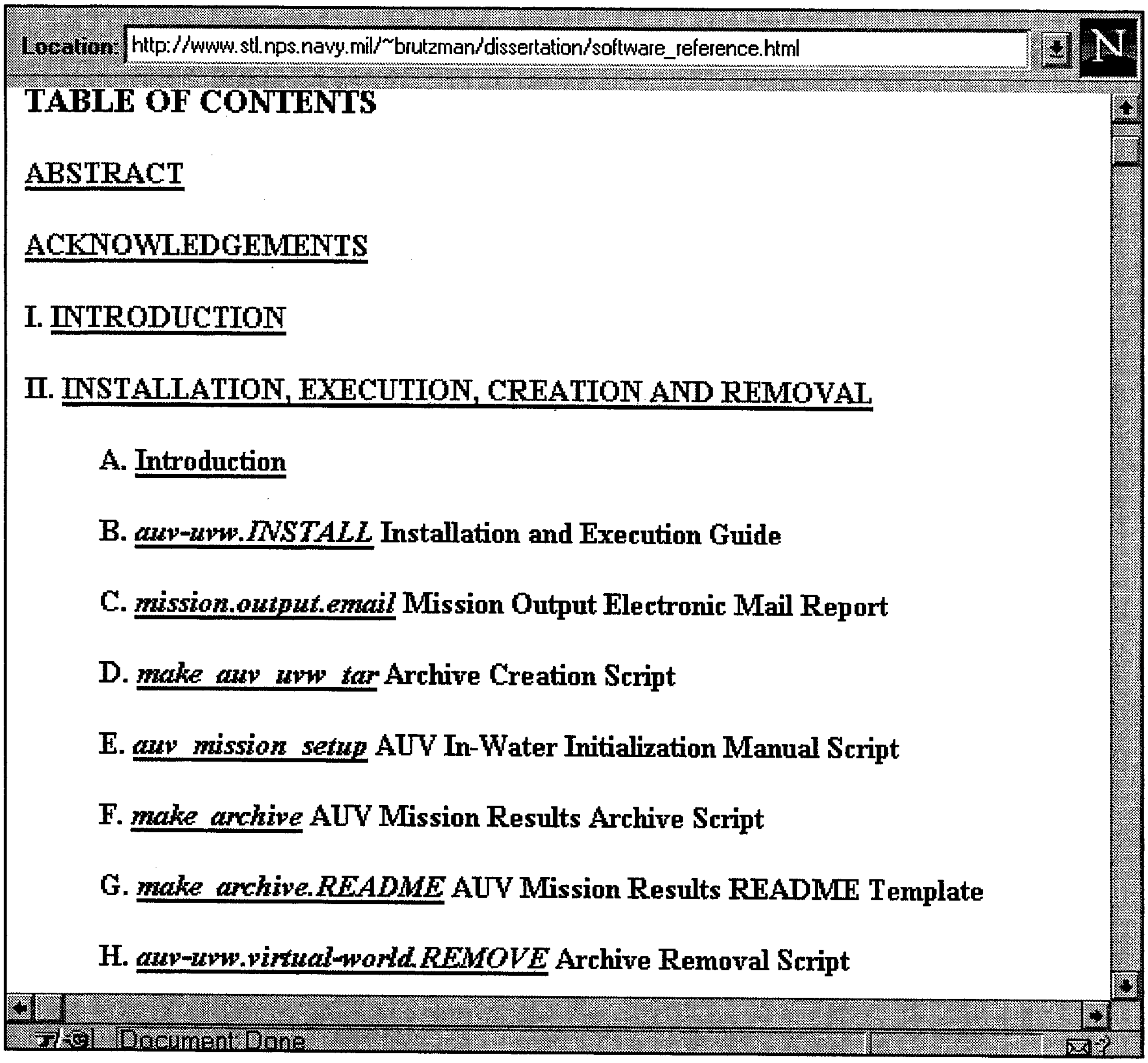

Location: http://www.stl.nps.navy.mil/~brutzman/dissertation/software_reference.html

TABLE OF CONTENTS

ABSTRACT

ACKNOWLEDGEMENTS

I. INTRODUCTION

II. INSTALLATION, EXECUTION, CREATION AND REMOVAL

A. Introduction

B. *auv-uvw.INSTALL* Installation and Execution Guide

C. *mission.output.email* Mission Output Electronic Mail Report

D. *make_auv_uvw_tar* Archive Creation Script

E. *auv_mission_setup* AUV In-Water Initialization Manual Script

F. *make_archive* AUV Mission Results Archive Script

G. *make_archive.README* AUV Mission Results README Template

H. *auv-uvw.virtual-world.REMOVE* Archive Removal Script

Document: Done

Figure 47. Software reference for downloading of the current version of software.

Running the TACTICALSTANDALONE version of the Tactical level code is accomplished by changing to the tactical directory from the main directory and typing, tactical. If the Prolog version is to be run the tactical code will be automatically available as part of the Make process. To run the Prolog code the current mission file needs to be called mission.pl. Prolog is started from the tactical directory where the Prolog code resides. Several missions are available with all of them starting with mission.pl.[descriptive extension]. Once Prolog is running, the input [mission]. is typed to load the mission file and the Tactical level code. To start the mission, the command execute_mission. is typed at the Prolog prompt. Once the mission starts, it automatically sets up network communications and awaits connection, which the Execution level performs.

APPENDIX F. MAKEFILE SOURCE CODE

The Makefile became increasing important as the number of files with dependencies grew. Below is the Makefile being used for the Tactical level.

```
################################################################################
#
# Program:  Makefile    AUV tactical level Makefile
#
# Author:   Don Brutzman, Brad Leonhardt, Dave McClarin
#
# Revised:  15 March 96
#
# System:   Irix 5.3
#
# Invocation:     make
#     or    make tactical
#     or    make strategic
#     or    make both
#     or    make all
#
# Notes:    Abandon hope all ye who enter here!
#
#           make all whenever you copy program files from somewhere else
#           since file dates are not synchronized across different systems.
#
################################################################################

FLAGS        = -cckr -g -lm -w

EPATH        = ../execution/

EXECUTION_CODE     = $(EPATH)globals.c       $(EPATH)statevector.c             \
              $(EPATH)parse_functions.c   $(EPATH)external_functions.c

EXECUTION_OBJS     = $(EPATH)globals.o       $(EPATH)statevector.o             \
             $(EPATH)parse_functions.o    $(EPATH)external_functions.o
```

```
EXECUTION_DEFS     = $(EPATH)globals.h     $(EPATH)statevector.h              \
                     $(EPATH)defines.h

TACTICAL_CODE      = tactical.c            replanner.c                        \
                     c_search.c            circle.c                           \
                     circtest.c            navigator1.c                       \
                     kalman_filter.c       readgps.c                          \
                     matrix.c              sonar.c                            \
                     sonar_comms.c

TACTICAL_OBJS      = tactical.o            replanner.o                        \
                     c_search.o            circle.o                           \
                     circtest.o            navigator1.o                       \
                     kalman_filter.o       readgps.o                          \
                     matrix.o              sonar.o                            \
                     sonar_comms.o

TACTICAL_DEFS      = kalman_filter.h       readgps.h                          \
                     matrix.h              dt2cl.h                            \
                    sonar_classification.h sonar_comms.h                      \
                     sonar_globals.h

#################################################################################

# first rule in Makefile is default

both:
      make tactical
      make strategic

tactical: $(EXECUTION_CODE) $(EXECUTION_OBJS) $(EXECUTION_DEFS)               \
         $(TACTICAL_CODE)  $(TACTICAL_OBJS)  $(TACTICAL_DEFS) Makefile
      @echo "Linking tactical..."
      cc    $(FLAGS) -DTACTICAL_STANDALONE                                    \
            $(EXECUTION_OBJS) $(TACTICAL_OBJS) -o tactical
      @echo "Make tactical complete."

strategic: $(EXECUTION_CODE) $(EXECUTION_OBJS) $(EXECUTION_DEFS)              \
         $(TACTICAL_CODE)  $(TACTICAL_OBJS)  $(TACTICAL_DEFS) Makefile
```

```
	cc	$(FLAGS) -DTACTICAL_STANDALONE=0 -c tactical.c			\
		-o tactical.o
	@echo "Linking strategic (requires Prolog compiler)..."
	ld -r -lsocket -lm -lnsl -z muldefs *.o -o tactical.so
	cp tactical.so tactical.o
	@echo "Make strategic complete."

##################################################################################

# object files

$(EPATH)globals.o: $(EPATH)globals.c $(EXECUTION_DEFS) Makefile
	cc	$(FLAGS) -c $(EPATH)globals.c					\
		-o $(EPATH)globals.o

$(EPATH)statevector.o: $(EPATH)statevector.c $(EXECUTION_DEFS) Makefile
	cc	$(FLAGS) -c $(EPATH)statevector.c				\
		-o $(EPATH)statevector.o

$(EPATH)parse_functions.o: $(EPATH)parse_functions.c $(EXECUTION_DEFS) Makefile
	cc	$(FLAGS) -c $(EPATH)parse_functions.c				\
		-o $(EPATH)parse_functions.o

$(EPATH)external_functions.o: $(EPATH)external_functions.c $(EXECUTION_DEFS)	\
		Makefile
	cc	$(FLAGS) -c $(EPATH)external_functions.c			\
		-o $(EPATH)external_functions.o

 tactical.o:	tactical.c $(TACTICAL_DEFS) Makefile
	cc	$(FLAGS) -DTACTICAL_STANDALONE -c tactical.c -o tactical.o

replanner.o:	replanner.c $(TACTICAL_DEFS) Makefile
	cc	$(FLAGS) -c replanner.c -o replanner.o

c_search.o: c_search.c $(TACTICAL_DEFS) Makefile
	cc	$(FLAGS) -c c_search.c -o c_search.o

circle.o:	circle.c $(TACTICAL_DEFS) Makefile
	cc	$(FLAGS) -c circle.c -o circle.o
```

```
circtest.o: circtest.c $(TACTICAL_DEFS) Makefile
	cc	$(FLAGS) -c circtest.c -o circtest.o

navigator1.o:	navigator1.c $(TACTICAL_DEFS) Makefile
	cc	$(FLAGS) -c navigator1.c -o navigator1.o

kalman_filter.o: kalman_filter.c $(TACTICAL_DEFS) Makefile
	cc	$(FLAGS) -c kalman_filter.c -o kalman_filter.o

readgps.o:	readgps.c $(TACTICAL_DEFS) Makefile
	cc	$(FLAGS) -c readgps.c -o readgps.o

matrix.o:	matrix.c $(TACTICAL_DEFS) Makefile
	cc	$(FLAGS) -c matrix.c -o matrix.o

sonar_comms.o:	sonar_comms.c $(TACTICAL_DEFS) Makefile
	cc	$(FLAGS) -c sonar_comms.c -o sonar_comms.o

sonar.o:	sonar.c $(TACTICAL_DEFS) Makefile
	cc	$(FLAGS) -c sonar.c -o sonar.o

################################################################################

# warnings, all, clean

# The 'warnings' make option gives voluminous diagnostics which are useful
# in preventing mysterious bugs when the execution code is ported to OS-9.
# Similar to lint.

warnings:
	cc	$(FLAGS) -fullwarn -wlint,p					\
		$(EXECUTION_CODE) $(TACTICAL_CODE) -o tactical

all:
	touch $(EXECUTION_CODE) $(EXECUTION_DEFS) $(TACTICAL_CODE) $(TACTICAL_DEFS)
	make both
clean:
	rm tactical *.o $(EXECUTION_OBJS)
################################################################################
```

LIST OF REFERENCES

Boorda, J. M., *Mine countermeasures - An Integral Part Of Our Strategy And Our Forces,* White Paper, December 1995.

Brutzman, Donald P., *From Virtual World to Reality: Designing an Autonomous Underwater Robot,* Proceedings of the Autonomous Vehicles in Mine Countermeasures Symposium, Naval Postgraduate School, Monterey CA, April 1995.

Brutzman, Donald P., *From Virtual World to Reality: Designing an Autonomous Underwater Robot,* Proceedings of the Autonomous Vehicles in Mine Countermeasures Symposium, Naval Postgraduate School, Monterey CA, April 1995.

Brutzman, Donald P., *NPS Phoenix AUV Software Integration and In-Water Testing,*, AUV 96, Monterey, California, June 1996.

Brutzman, Donald P., *NPS AUV Integrated Simulator*, Master's Thesis, Naval Postgraduate School, Monterey, CA, March 1992.

Burns, Michael., *An Experimental Evaluation and Modification of Simulator-based Vehicle Control Software for the Phoenix Autonomous Underwater Vehicle (AUV),* Master's Thesis, Naval Postgraduate School, Monterey, CA, April 1996.
Available at *http://www.cs.nps.navy.mil/research/auv*

Byrnes, R.B., *The Rational Behavior Software Architecture for Intelligent Ships, An Approach to Mission and Motion Control,* Naval Engineers Journal, March 1996.

Byrnes, R.B., *The Rational Behavior Model: A Multi-Paradigm, Tri-Level Software Architecture for the Control of Autonomous Vehicles,* PH.D. Dissertation, Naval Postgraduate School, Monterey, CA, March 1993.

Campbell, Michael, *Real-Time Sonar Classification for Autonomous Underwater Vehicles*, Master's Thesis, Naval Postgraduate School, Monterey, CA, March 1996.
Available at *http://www.cs.nps.navy.mil/research/auv*

Davis, Duane & Leonhardt, Bradley J., *Class project CS4310 Expert System Graphical User Interface for Mission Autocode Generation,* Naval Postgraduate School, Monterey, CA, June 1995.

Davis, Duane, *Precision Maneuvering of Phoenix AUV into a Tunnel; Virtual World Verification and Application,* Master's Thesis, Naval Postgraduate School, Monterey, CA, September 1996.

Floyd, Charles A., *Design and Implementation of a Collision Avoidance System for the NPS Autonomous Underwater Vehicle (AUVII) Utilizing Ultrasonic Sensors,* Master's Thesis, Naval Postgraduate School, Monterey, CA, September 1991.

Gonzalez, Avelino J. *The Engineering of Knowledge-Based Systems Theory and Practice,* Prentice Hall, Inc. 1993.

Healey, A.J., *Mission Planning, Execution, and Data Analysis for the NPS AUV II Autonomous Underwater Vehicle,* Proceedings of the First IARP Workshop on Mobile Robots for Subsea Environments, Monterey, California, October, 1990

Healey, A.J., *Research on Autonomous Vehicles at the Naval Postgraduate School,* Naval Research Reviews, Office of Naval Research, Washington, D.C., Volume XLIV, Number 1, Spring 1992.

Holden Michael J., *Ada Implementation of Concurrent Execution for Multiple Tasks in the Strategic and Tactical Levels of the Rational Behavior Model for the NPS AUV,* Master's Thesis, Naval Postgraduate School, Monterey, CA, September 1995.

Kanayama, Y., *CS4313: Lecture Notes Introduction to Motion Planning*, Naval Postgraduate School, Monterey CA, March 1995.

Lozano-Pérez, *An Algorithm for Planning Collision-Free Paths among Polyhedral Obstacles,* Communications of the ACM Vol. 22 no. 10, October 1979.

Leonhardt, Bradley J., *Class project CS4314 Lisp Simulation of the OOD Module,* Naval Postgraduate School, Monterey, CA, June 1995.

Marco, David, *Autonomous Underwater Vehicles: Hybrid Control of Mission and Motion,* Journal of Autonomous Robots, 1996.

McClarin, David, *Kalman-Filtering of Navigation Data for the Phoenix Autonomous Underwater Vehicle*, Master's Thesis, Naval Postgraduate School, Monterey, CA, March 1996.

McGhee, Robert B., *CS4314: Lecture Notes Symbolic Computing*, Naval Postgraduate School, Monterey, CA, June 1995.

Oram, Andrew, *Managing Projects with Make,* O'Reilly and Associates, Inc., 1991.

Quintus Prolog, Quintus Corporation, Palo Alto CA, 1991.

Quintus Prowindows User's Guide, Quintus Computer Systems, Inc., Mountain View CA, 1988.

Rowe, Neil. C., *Artificial Intelligence Through Prolog,* Prentice-Hall, Inc., 1988.

Scrivener, Arthur, *Acoustic Underwater Navigation of the Autonomous Underwater Vehicle using the DiveTracker System*, Master's Thesis, Naval Postgraduate School, Monterey, CA, March 1996.

Wind River Systems, *VxWorks Programmer's Guide 5.2*, Wind River Systems Inc. March 1995.

Stevens, Richard W., *Advanced Programming in the UNIX Environment,* Addison-Wesley Publishing Company, 1992.

Tanenbaum, Andrew S., *Modern Operating Systems*, Prentice Hall, Inc., 1992.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center. 2
   Cameron Station
   Alexandria, VA 22304-6145

2. Library, Code 013. 2
   Naval Postgraduate School
   Monterey, CA 93943-5101

3. Computer Technology Programs, Code CS 1
   Naval Postgraduate School
   Monterey, CA 93943-5000

4. Dr. Ted Lewis, Code CS/Lt 1
   Chair, Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5100

5. Dr. Robert McGhee, Code CS/Mz 2
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5100

6. Dr. Donald P. Brutzman, Code UW/Br 2
   Undersea Warfare Academic Group
   Naval Postgraduate School
   Monterey, CA 93943-5100

7. Dr. Anthony J. Healey, Code ME/Hy 1
   Mechanical Engineering Department
   Naval Postgraduate School
   Monterey CA, 93943-5100

8. CDR Michael J. Holden, USN, Code CS/Hm. 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5100

9. LT Bradley J. Leonhardt 2
   1628 N. Owaissa St.
   Appleton WI 54911

10. David Marco, Code ME/MA 1
Mechanical Engineering Department
Naval Postgraduate School
Monterey, California 93943-5000

11. Russell Whalen, Code CS/ 1
Computer Science Department
Naval Postgraduate School
Monterey, California 93943-5000

12. Mr. Norman Caplan 1
National Science Foundation
BES, Room 565
4201 Wilson Blvd.
Arlington, VA 22230

13. Dr. James Bellingham 1
Underwater Vehicles Laboratory, MIT Sea Grant College Program
292 Main Street
Massachusetts Institute of Technology
Cambridge Massachusetts 02142